Handbuch

zum

PC-volksFORTH83

(rev. 3.81)

Handbuch zum PC-volksFORTH83 rev. 3.81

Die Autoren haben sich in diesem Handbuch um eine vollständige und akkurate Darstellung bemüht. Die im Handbuch enthaltenen Informationen dienen jedoch allein der Produktbeschreibung und sind nicht als zugesicherte Eigenschaften im Rechtssinne aufzufassen. Etwaige Schadensersatzansprüche gegen die Autoren gleich aus welchem Rechtsgrund, sind ausgeschlossen. Es wird keine Gewähr übernommen, daß die angegebenen Verfahren frei von Schutzrechten Dritter sind.

Alle Rechte vorbehalten. Ein Nachdruck, auch auszugsweise, ist nur zulässig mit Einwilligung der Autoren und genauer Quellenangabe sowie Einsendung eines Belegexemplars an die Autoren.

(c) 1985,1986 Bernd Pennemann, Georg Rehfeld, Dietrich Weineck
(c) 1988,1989 Klaus Schleisiek, Jörg Staben, Klaus Kohl

- Mitglieder der FORTH Gesellschaft e.V. -

Unser Dank gilt der gesamten FORTH-Gemeinschaft,
insbesondere Charles Moore, Michael Perry und Henry Laxen

1. Auflage Mai 1989

Inhaltsverzeichnis

1. Prolog 6
1.1 Interpreter und Compiler 6
1.2 Warum stellen wir dieses System frei zur Verfügung ? 9
1.3 Warum soll man in volksFORTH83 programmieren ? 9
1.4 Hinweise des Lektors 11

2. Einstieg ins volksFORTH 12
2.1 Die Systemdiskette 12
2.2 Die Oberfläche 14
2.3 Arbeiten mit Programm- und Datenfiles 15
2.4 Der Editor 16
2.4.1 HELP und VIEW 16
2.4.2 Öffnen und Editieren eines Files 17
2.4.3 Tastenbelegung des Editors 18
2.4.4 Beispiel: CLS 19
2.4.5 Compilieren im Editor 21
2.5 Erstellen einer Applikation 22
2.6 Das Erstellen eines eigenen FORTH-Systems 23
2.7 Ausdrucken von Screens 25
2.8 Druckeranpassung 26

3. Arithmetik 28
3.1 Stacknotation 28
3.2 Arithmetische Funktionen 29
3.3 Logik und Vergleiche 32
3.4 32Bit-Worte 33
3.5 Stack-Operationen 35
3.5.1 Datenstack-Operationen 36
3.5.2 Returnstack-Operationen 38

4. Kontrollstrukturen 40
4.1 Programm-Strukturen 40
4.2 Worte zur Fehlerbehandlung 46
4.3 Fallunterscheidung in FORTH 47
4.3.1 Strukturierung mit IF ELSE THEN / ENDIF 47
4.3.2 Behandlung einer CASE - Situation 50
4.3.2.1 Strukturelles CASE 50
4.3.2.2 Positionelles CASE 54
4.3.2.3 Einsatzmöglichkeiten 57
4.4 Rekursion 59

5. Ein- / Ausgabe im volksFORTH 61
5.1 Ein- / Ausgabebefehle im volksFORTH 61
5.2 Ein- / Ausgaben über Terminal 62
5.3 Drucker-Ausgaben 66
5.4 Ein- / Ausgabe von Zahlen 66
5.5 Ein- / Ausgabe über einen Port 67
5.6 Eingabe von Zeichen 67
5.7 Ausgabe von Zeichen 73

6. Strings 75
6.1 String-Manipulationen 76
6.2 Suche nach Strings 79
6.2.1 In normalem Fließtext 79
6.2.2 Im Dictionary 79
6.3 Ø-terminated Strings 8Ø
6.4 Konvertierungen: Strings -- Zahlen 81
6.4.1 String in Zahlen wandeln 81
6.4.2 Zahlen in Strings wandeln 83

7. Umgang mit Dateien 84
7.1 Pfad-Unterstützung 89
7.2 DOS-Befehle 89
7.3 Block-orientierte Dateien 9Ø
7.4 Verwaltung der Block-Puffer 97
7.5 Index-, Verschiebe- und Kopierfunktionen für Block-Files 99
7.6 FCB-orientierte Dateien 1ØØ
7.7 HANDLE-orientierte Dateien 1Ø3
7.8 Direkt-Zugriff auf Disketten 1Ø5
7.9 Fehlerbehandlung 1Ø5

8. Speicheroperationen 1Ø7
8.1 Speicheroperationen im 16-Bit-Adressraum 1Ø7
8.2 Segmentierte Speicheroperationen 111

9. Datentypen 113
9.1 Ein- und zweidimensionale Felder 118
9.2 Methoden der objektorientierte Programmierung 121

1Ø. Manipulieren des Systemverhalten 123
1Ø.1 Patchen von FORTH-Befehlen 123
1Ø.2 Verwendung von DEFER-Wörtern 124
1Ø.3 Neudefinition von Befehlen 125
1Ø.4 DEFERed Wörter im volksFORTH83 126
1Ø.5 Vektoren im volksFORTH83 127
1Ø.6 Glossar 128

11. Vokabular-Struktur 132
11.1 Die Suchreihenfolge 132
11.2 Glossar 133

12. Dictionary-Struktur 136
12.1 Aufbau 136
12.2 Glossar 140

13. Der HEAP 144

14. Die Ausführung von FORTH-Worten 146
14.1 Der Aufbau des Adressinterpreters 146
14.2 Die Funktion des Adressinterpreters 146
14.3 Verschiedene IMMEDIATE-Worte 148
14.4 Die Does>-Struktur 149
14.5 Glossar 150

15. Der Assembler 158
15.1 Registerbelegung 158
15.2 Registeroperationen 159
15.3 Besonderheiten im volksFORTH83 159
15.4 Glossar 160
15.5 Kontrollstrukturen im Assembler 162
15.6 Beispiele aus dem volksFORTH 162

16. Der Multitasker 165
16.1 Implementation 166
16.2 Semaphore und Lock 167
16.3 Glossar 169

17. Debugging-Techniken 174
17.1 Der Tracer 174
17.2 Debug 178
17.2.1 Beispiel: EXAMPLE 178
17.2.2 NEST und UNNEST 178
17.3 Stacksicherheit 180
17.4 Aufrufgeschichte 182
17.5 Dump 182
17.6 Dekompiler 183
17.7 Glossar 185

18. Begriffe 186
18.1 Entscheidungskriterien 186
18.2 Definition der Begriffe 186

Indexverzeichnis 196

1. Prolog

volksFORTH83 ist eine Sprache, die in verschiedener Hinsicht ungewöhnlich ist.
Denn FORTH selbst ist nicht nur eine Sprache, sondern ein im Prinzip grenzenloses Programmiersystem. Eines der Hauptmerkmale des Programmiersystems FORTH ist seine Modularität. Diese Modularität wird durch die kleinste Einheit eines FORTH-Systems, das **WORT**, gewährleistet.
In FORTH werden die Begriffe Prozedur, Routine, Programm, Definition und Befehl alle gleichbedeutend mit Wort gebraucht. FORTH besteht also, wie jede andere natürliche Sprache auch, aus Wörtern.

Diese FORTH-Worte kann man als bereits kompilierte Module betrachten, wobei immer ein Kern aus einigen hundert Worten durch eigene Worte erweitert wird. Diese Worte des Kerns sind in einem FORTH83-Standard festgelegt und stellen sicher, daß Standard-Programme ohne Änderungen auf dem jeweiligen FORTH-System lauffähig sind.
Ungewöhnlich ist, daß der Programmtext des Kerns selbst ein FORTH-Programm ist, im Gegensatz zu anderen Programmiersprachen, denen ein Maschinensprach-Programm zugrunde liegt. Aus diesem Kern wird durch ein besonderes FORTH-Programm, den MetaCompiler, das lauffähige KERNEL.COM erzeugt:

Wie fügt man nun diesem lauffähigen Kern eigene Worte hinzu?
Das Kompilieren der Worte wird in FORTH mit COLON ":" eingeleitet und mit SEMICOLON ";" abgeschlossen:

: erwartet bei der Ausführung einen Namen und ordnet diesem Namen alle nachfolgenden Wörter zu.

; beendet diese Zuweisung von Wörtern an den Namen und stellt das neue Wort unter diesem Namen für Aufrufe bereit.

1.1 Interpreter und Compiler

Ein klassisches FORTH-System stellt immer sowohl einen Interpreter als auch einen Compiler zur Verfügung. Nach der Einschaltmeldung oder einem Druck auf die <CR>-Taste wartet der FORTH-Interpreter mit dem FORTH-typischen "ok" auf Ihre Ein-

gabe. Sie können ein oder mehrere Befehlswörter in eine Zeile schreiben. volksFORTH beginnt erst nach einem Druck auf die RETURN-Taste <CR> mit seiner Arbeit, indem es der Reihe nach jeden Befehl in der Eingabezeile abarbeitet. Dabei werden Befehlsworte durch Leerzeichen begrenzt. Der Delimiter (Begrenzer) für FORTH-Prozeduren ist also das Leerzeichen, womit auch schon die Syntax der Sprache FORTH beschrieben wäre.

Der Compiler eines FORTH-Systems ist also Teil der Interpreteroberfläche. Es gibt daher keinen Compiler-Lauf zur Erstellen des Programmtextes wie in anderen Compiler-Sprachen, sondern der Interpreter wird mit allen zur Problemlösung notwendigen Worten als Anwenderprogramm abgespeichert.

Auch : (COLON) und ; (SEMICOLON) sind kompilierte Worte, die aber für das System den Compiler ein- und ausschalten. Da sogar die Worte, die den Compiler steuern, "normale" FORTH-Worte sind, fehlen in FORTH die in anderen Sprachen üblichen Compiler-Optionen oder Compiler-Schalter. Der FORTH-Compiler wird mit FORTH-Worten gesteuert.

Der Aufruf eines FORTH-Wortes erfolgt über seinen Namen ohne ein explizites CALL oder GOSUB. Dies führt zum FORTH-typischen Aussehen der Wortdefinitionen:

```
: <name>
    <wort1> <wort2> <wort3> ... ;
```

Die Standard-Systemantwort in FORTH ist das berühmte "ok". Ein Anforderungszeichen wie 'A>' bei DOS oder ']' beim guten APPLE II gibt es nicht! Das kann dazu führen, daß nach einer erfolgreichen Aktion der Bildschirm völlig leer bleibt; getreu der Devise:

Keine Nachrichten sind immer gute Nachrichten !

Und - ungewöhnlicherweise - benutzt FORTH die sogenannte Postfix notation (UPN) vergleichbar den HP-Taschenrechnern, die in machen Kreisen sehr beliebt sind. Das bedeutet, FORTH erwartet immer erst die Argumente, dann die Aktion. Statt

```
                    3 + 2   und   (5 + 5) * 10
heißt es            2 3 +   und   5 5 + 10 *  .
```

Da die Ausdrücke von links nach rechts ausgewertet werden, gibt es in FORTH keine Klammern.

Stack

Ebenso ungewöhnlich ist, daß FORTH nur ausdrücklich angeforderte Aktionen ausführt: Das Ergebnis Ihrer Berechnungen bleibt solange in einem speziellen Speicherbereich, dem Stack, bis es mit einem Ausgabebefehl (meist .) auf den Bildschirm oder dem Drucker ausgegeben wird.
Da die FORTH-Worte den Unterprogrammen und Funktionen anderer Programmiersprachen entsprechen, benötigen sie gleichfalls die Möglichkeit, Daten zur Verarbeitung zu übernehmen und das Ergebnis abzulegen. Diese Funktion übernimmt der

STACK. In FORTH werden Parameter für Prozeduren selten in Variablen abgelegt, sondern meist über den Stack übergeben.

Assembler

Innerhalb einer FORTH-Umgebung kann man sofort in der Maschinensprache des Prozessors programmieren, ohne den Interpreter verlassen zu müssen. Assembler-Definitionen sind den FORTH-Programmen gleichwertige FORTH-Worte.

Vokabular-Konzept

Das volksFORTH verfügt über eine erweiterte Vokabular-Struktur, die von W. Ragsdale vorgeschlagen wurde. Dieses Vokabular-Konzept erlaubt das Einordnen der FORTH-Worte in logische Gruppen.
Damit können Sie notwendige Befehle bei Bedarf zuschalten und nach Gebrauch wieder wegschalten. Darüberhinaus ermöglichen die Vokabulare den Gebrauch gleicher Namen für verschiedene Worte, ohne in einen Namenskonflikt zu kommen. Eine im Ansatz ähnliche Vorgehensweise bietet das UNIT-Konzept moderner PASCAL- oder MODULA-Compiler.

FORTH-Dateien

FORTH verwendet oftmals besondere Dateien für seine Programme. Dies ist historisch begründet und das Erbe einer Zeit, als FORTH noch sehr oft Aufgaben des Betriebssystems übernahm. Da gab es ausschließlich FORTH-Systeme, die den Massenspeicher vollständig selbst ohne ein DOS verwalteten und dafür ihre eigenen Dateistrukturen benutzten.
Diese Dateien sind sogenannte Blockfiles und bestehen aus einer Aneinanderreihung von 1Ø24 Byte großen Blöcken. Ein solcher Block, der gerne **SCREEN** genannt wird, ist die Grundlage der Quelltext-Bearbeitung in FORTH.
Allerdings können mit dem volks4TH auch Dateien bearbeitet werden, die im Dateiformat des MS-DOS vorliegen, sog. "Stream Files".

Generell steht hinter jeder Sprache ein bestimmtes Konzept, und nur mit Kenntnis dieses Konzeptes ist möglich, eine Sprache effizient einzusetzen. Das Sprachkonzept von FORTH wird beschrieben in dem Buch "In FORTH denken" von Leo Brodie (Hanser Verlag).

Einen ersten Eindruck vom volksFORTH83 und von unserem Stolz darüber soll dieser Prolog vermitteln. volksFORTH83 ist ein "Public-Domain"-System, bei dessen Leistungsfähigkeit sich die Frage stellt:

1.2 Warum stellen wir dieses System frei zur Verfügung ?

Die Verbreitung, die die Sprache FORTH gefunden hat, war wesentlich an die Existenz von figFORTH geknüpft. Auch figFORTH ist ein public domain Programm, d.h. es darf weitergegeben und kopiert werden. Trotzdem haben sich bedauerlicherweise verschiedene Anbieter die einfache Adaption des figFORTH an verschiedene Rechner sehr teuer bezahlen lassen.

Das im Jahr 1979 erschienene figFORTH ist heute nicht mehr so aktuell, weil mit der weiteren Verbreitung von Forth eine Fülle von eleganten Konzepten entstanden sind, die z.T. in den Forth83-Standard Eingang gefunden haben. Daraufhin wurde von Laxen und Perry das F83 geschrieben und als Public Domain verbreitet. Dieses freie 83er-Standard-FORTH mit seinen zahlreichen Utilities ist recht komplex und wird auch nicht mit Handbuch geliefert.

Wir haben ein neues Forth für verschiedene Rechner entwickelt. Das Ergebnis ist das volksFORTH83, eines der besten Forth-Systeme, die es gibt. Das volksFORTH soll an die Tradition der oben genannten Systeme, insbesondere des F83, anknüpfen und die Verbreitung der Sprache FORTH fördern.

volksFORTH wurde unter dem Namen ultraFORTH zunächst für den C64 geschrieben. Nach Erscheinen der Rechner der Atari ST-Serie entschlossen wir uns, auch für sie ein volksFORTH83 zu entwickeln. Die erste ausgelieferte Version 3.7 war, was Editor und Massenspeicher betraf, noch stark an den C64 angelehnt. Sie enthielt jedoch schon einen verbesserten Tracer, die GEM-Bibliothek und die anderen Tools für den ST. Der nächste Schritt bestand in der Einbindung der Betriebssystem-Files. Nun konnten Quelltexte auch vom Desktop und mit anderen Utilities verarbeitet werden. Die dritte Adaption des volksFORTH entstand für die CP/M-Rechner (8Ø8Ø-Prozessoren), wobei speziell für den Schneider CPC auch die Grafikfähigkeit unterstützt wird. Zuletzt wurde das volksFORTH für die weitverbreiteten Rechner der IBM PC-Serie angepaßt. Mit der jetzt ausgelieferten Version 3.81 ist das volksFORTH vollständig.

1.3 Warum soll man in volksFORTH83 programmieren ?

Das volksFORTH83 ist ein ausgesprochen leistungsfähiges und kompaktes Werkzeug.

Durch residente Runtime-library, Compiler, Editor und Debugger sind die ermüdenden ECLG-Zyklen ("Edit, Compile, Link and Go") überflüssig. Der Code wird Modul für Modul entwickelt, kompiliert und getestet.

Der integrierte Debugger ist die perfekte Testumgebung für Worte. Es gibt keine riesigen Hexdumps oder Assemblerlistings, die kaum Ähnlichkeit mit dem Quelltext haben.

Ein anderer wichtiger Aspekt ist das Multitasking. So wie man ein Programm in einzelne, unabhängige Module oder Worte aufteilt, so sollte man es auch in ein-

zelne, unabhängige Prozesse aufteilen können. Das ist in den meisten Sprachen nicht möglich. Das volksFORTH83 besitzt einen einfachen, aber leistungsfähigen Multitasker.

Schließlich besitzt das volksFORTH83 noch eine Fülle von Details, über die andere FORTH-Systeme nicht verfügen:

- Es benutzt an vielen Stellen Vektoren und sog. deferred Worte, die eine einfache Umgestaltung des Systems für verschiedene Gerätekonfigurationen ermöglichen.
- Es besitzt einen Heap für "namenlose" Worte oder für Code, der nur zeitweilig benötigt wird.
- Der Blockmechanismus ist so schnell, daß er auch sinnvoll für die Bearbeitung großer Datenmengen, die in Files vorliegen, eingesetzt werden kann.
- Das System umfaßt Tracer, Decompiler, Multitasker, Assembler, Editor, Printer-Interface ...

Das volksFORTH83 erzeugt, verglichen mit anderen FORTH-Systemen, relativ schnellen Code, der aber langsamer als der anderer Compilersprachen ist.

Mit diesem Handbuch soll die Unterstützung des volksFORTH83 noch nicht Zuende sein. Die FORTH Gesellschaft e.V., ein gemeinnütziger Verein, bietet dafür die Plattform. Sie gibt die Vereins-FORTH-Zeitschrift "VIERTE DIMENSION" heraus und betreibt den FORTH-Tree, eine ungewöhnliche, aber sehr leistungsfähige Mailbox.

Die im Frühsommer 1989 aktuelle Adressen:

FORTH-Büro:

FORTH Gesellschaft e.V.
Postfach 1110
8044 Unterschleißheim
Tel. 089/3173784

volksFORTH-Vertrieb:

Michael & Klaus Kohl
Pestalozzistr. 69
8905 Mering
Tel. 08233/30524

1.4 Hinweise des Lektors

Diesem Handbuch zum PC-volksFORTH83 ist sowohl als Nachschlagewerk als auch als Lehrbuch für FORTH (speziell volksFORTH) gedacht. Deshalb handelt es sich nicht, wie bei den anderen volksFORTH-Handbücher, um eine Auflistung des Vokabulars. Statt dessen wird mit ausführlichen Beschreibungen und Programmbeispielen in vielen Kapiteln die Möglichkeiten des FORTH-Systems erklärt. Ergänzt werden die einzelnen Kapitel jeweils um Wortbeschreibungen der darin vorkommenden Befehle (Glossar). Sollten bestimmte Befehle gesucht werden, so ist die Seitennummer aus dem ausführlichen Index zu entnehmen.

Zur Unterscheidung von Beschreibung, FORTH-Worten, Programm-Eingaben und -ausgaben wird mit unterschriftlichen Schrifttypen gearbeitet:

Beschreibungen	erfolgen in Proportionalschrift mit Randausgleich.
FORTH-Befehle	werden im Text durch Fettschrift hervorgehoben.
Eingaben	und
Programmlistings	verwenden die nichtproportionale Schriftart Courier.
<u>Ausgaben</u>	des FORTH-Interpreter/Compiler sind unterstrichen.

2. Einstieg ins volksFORTH

Damit Sie sofort beginnen können, wird in diesem Kapitel beschrieben,
- wie man das System startet
- wie man sich im System zurechtfindet
- wie man ein fertiges Anwendungsprogramm erstellt
- wie man ein eigenes Arbeitssystem zusammenstellt

2.1 Die Systemdiskette

Zu Ihrem Handbuch haben Sie eine Diskette erhalten. Fertigen Sie auf jeden Fall mit dem DOS-Befehl **diskcopy** eine Sicherheitskopie dieser Diskette an. Die Gefahr eines Datenverlustes ist groß, da FORTH Ihnen in jeder Hinsicht freie Hand läßt - auch beim versehentlichen Löschen Ihrer Systemdisketten !!

Die Diskette, auf der Ihr volksFORTH-System ausgeliefert wird, enthält folgende Dateien:

INSTALL	BAT	ist ein Installationsprogramm, das volks4TH auf einem angegebenen Laufwerk einrichtet.
PKXARC	COM	ist ein Dienstprogramm zum Komprimieren und Dekomprimieren von Dateien.
KERNEL	COM	
MINIMAL	COM	
VOLKS4TH	COM	diese drei COM-Files sind drei volksFORTH-Systeme in verschiedenen Ausbaustufen.
FORTH1	ARC	
FORTH2	ARC	diese beiden ARC-Dateien enthalten die Quelltexte des gesamten FORTH-Systems, müssen aber erst von PKXARC entpackt werden.
VOLKS4TH	DOC	ist eine ergänzende Dokumentation, die Nachträge enthält.
READ	ME	enthält zusätzliche wichtige Hinweise.

Wenn Sie Ihr System, wie in der Datei READ.ME auf der Diskette beschrieben, idealerweise auf einer Festplatte installiert haben, finden Sie deutlich mehr Files

vor. Ein Zeichen für die Platzersparnis durch das Datei-Kompressionsprogramm PKARC.

VOLKS4TH	COM	als Ihr komplettes Arbeitsssystem enthält resident das Fileinterface, den Editor, den Assembler und von Ihnen eingefügte Werkzeuge (tools).
MINIMAL	COM	ist eine Grundversion, die oft benötigte Systemteile enthält. Diese ist notwendig, da FORTH-Systeme allgemein nicht über einen Linker verfügen, sondern ausgehend vom Systemkern die zur Problemlösung notwendigen Einzelprogramme schrittweise hinzukompiliert werden.
KERNEL	COM	ist eine Grundversion, die nur den Sprachkern enthält. Damit können Sie eigene FORTH-Versionen mit z.B. einem veränderten Editor zusammenstellen und dann mit **SAVESYSTEM <name>** als fertiges System abspeichern. In der gleichen Art können Sie auch fertige Applikationen herstellen, denen man ihre FORTH-Abstammung nicht mehr ansieht.
KERNEL	SCR	enthält die Quelltexte des Sprachkerns. Eben dieser Quelltext ist mit einem Target-Compiler kompiliert worden und entspricht exakt dem KERNEL.COM. Sie können sich also den Compiler ansehen, wenn Sie wissen wollen, wie das volksFORTH83 funktioniert !
VOLKS4TH	SYS	enthält einen Ladeblock (Block 1), der alle Teile kompiliert, die zu Ihrem Arbeitssystem gehören. Mit diesem Loadscreen ist aus KERNEL.COM das File VOLKS4TH.COM zusammengestellt worden.
EXTEND	SCR	enthält Erweiterungen des Systems. Hier tragen Sie auch persönliche Erweiterungen ein.
CED	SCR	enthält den Quelltext des Kommandozeilen Editors, mit dem die Kommandozeile des Interpreters editiert werden kann. Soll dieser CED ins System eingefügt werden, so ist diese Datei mit **include ced.scr** **savesystem volks4TH.com** ins volksFORTH einzukompilieren.
HISTORY		wird von CED angelegt und enthält die zuletzt eingegebenen Kommandos.
STREAM	SCR	enthält zwei oft gewünschte Dienstprogramme: Die Umwandlung von Text-Dateien (stream files) in Block-Dateien (block files) und zurück.

DISASM	SCR	enthält den Dis-Assembler, der - wie beim CED beschrieben - ins System eingebaut werden kann.

2.2 Die Oberfläche

Wenn Sie VOLKS4TH von der DOS-Ebene starten, meldet sich volksFORTH83 mit einer Einschaltmeldung, die die Versionsnummer rev. <xxxx> enthält.

Was Sie nun von volksFORTH sehen, ist die Oberfläche des Interpreters. FORTH-Systeme und damit auch volksFORTH sind fast immer interaktive Systeme, in denen Sie einen gerade entwickelten Gedankengang sofort überprüfen und verwirklichen können. Das Auffälligste an der volksFORTH-Oberfläche ist die inverse Statuszeile in der unteren Bildschirmzeile, die sich mit **status off** aus- und mit **status on** wieder einschalten läßt.

Diese Statuszeile zeigt von links nach rechts folgende Informationen, wobei ¦ für "oder" steht:

<2¦8¦1Ø¦16>	die zur Zeit gültige Zahlenbasis (dezimal)
s <xx>	nennt die Anzahl der Zahlenwerte, die zum Verarbeiten bereitliegen
Dic <xxxx>	nennt den freien Speicherplatz
Scr <xx>	ist die Nummer des aktuellen Quelltextblocks
A:¦C:	gibt das aktuelle Laufwerk an
<name>.<ext>	zeigt den Namen der Date, die gerade bearbeitet wird. Dateien haben im MSDOS sowohl einen Namen <name> als auch eine dreibuchstabige Kennung, die Extension <ext>, wobei auch Dateien ohne Extension angelegt werden können.
FORTH FORTH FORTH	zeigt die aktuelle Suchreihenfolge gemäß dem Vokabularkonzept. Ein Beispiel dafür sind die Assembler-Befehle: Diese befinden sich in einem Vokabular namens **ASSEMBLER** und **assembler words** zeigt Ihnen den Befehlsvorrat des Assemblers an. Achten Sie bitte auf die rechte Seite der Statuszeile, wo jetzt **assembler forth forth** zu sehen ist. Da Sie aber jetzt - noch - keine Assembler-Befehle einsetzen wollen, schalten Sie bitte mit **forth** die Suchlaufpriorität wieder um. Die Statuszeile zeigt wieder das gewohnte **forth forth forth.**

Zur Orientierung im Arbeitssystem stellt das volksFORTH einige standardkonforme Wörter zur Verfügung:

words zeigt Ihnen die Befehlsliste von FORTH, die verfügbaren Wörter. Diese Liste stoppt bei einem Tastendruck mit der Ausgabe oder bricht bei einem <ESC> ab.

files zeigt alle im System angelegten logischen Datei-Variablen, die zugehörigen handle Nummern, Datum und Uhrzeit des letzten Zugriffs und ihre entsprechenden physikalischen DOS-Dateien. Eine solche FORTH-Datei wird allein durch die Nennung ihres Namens angemeldet. Die MSDOS-Dateien im Directory werden mit **dir** angezeigt.

path informiert über eine vollständige Pfadunterstützung nach dem MSDOS-Prinzip, allerdings vollkommen unabhängig davon.
Ist kein Suchpfad gesetzt, so gibt path nichts aus.

order beschreibt die Suchreihenfolge in den Befehlsverzeichnissen (Vokabular).

vocs nennt alle diese Unterverzeichnisse (vocabularies).

2.3 Arbeiten mit Programm- und Datenfiles

Um überhaupt Programmtexte (Quelltexte) schreiben zu können, brauchen Sie eine Datei, die diese Programmtexte aufnimmt. Diese Datei muß zur Bearbeitung angemeldet werden.

volksFORTH geht nach dem Systemstart erst einmal von der Datei VOLKS4TH.SYS als aktueller Datei aus. VOLKS4TH.SYS ist aber die Steuerdatei, aus der Ihr FORTH-System aufgebaut wurde; deshalb deklarieren Sie die Datei, die Sie bearbeiten wollen, mit dieser Befehlsfolge:

```
use <name>.<ext>
```

Als Beispiel wird die Datei test.scr mit `use test.scr` angemeldet.
Möchten Sie allerdings eine vollkommen neue Datei für Ihre Programme benutzen, so überlegen Sie sich einen Namen <name>, eine Kennung <extension> und eine vernünftige Größe in KByte. Anschließend geben Sie ein:

```
makefile <name>.<ext>  <Größenangabe> more
```

Daraufhin wird die Datei auf dem Laufwerk angelegt und zum Bearbeiten angemeldet. Ein **use** ist danach nicht mehr notwendig.

2.4 Der Editor

Zum Bearbeiten von Quelltext-Blöcken enthalten ältere FORTH-Systeme meist Editoren, die diesen Namen höchstens zu einer Zeit verdient haben, als man noch die Bits einzeln mit Lochzange und Streifenleser an die Hardware übermitteln mußte. Demgegenüber bot ein Editor, mit dem man jeweils eine ganze Zeile bearbeiten kann, sicher schon einigen Komfort. Da man jedoch mit solch einem Zeileneditor heute keinen Staat mehr machen und erst recht nicht mit anderen Sprachen konkurrieren kann, können Sie im volksFORTH selbstverständlich mit einem komfortablen Fullscreen-Editor arbeiten.

In diesem Editor kann man zwei Dateien gleichzeitig bearbeiten:
Eine Vordergrund-Datei, das aktuelle isfile und ein Hintergrundfile fromfile . Daher werden im Editor zwei Dateinamen angezeigt. Das Wort use meldet eine Datei automatisch sowohl als isfile als auch als fromfile an, so daß sich Verschiebe- und Kopieroperationen nur auf diese eine Datei beziehen.

Im Editor wird immer ein FORTH-Screen - also 1Ø24 Bytes - in der üblichen Aufteilung in 16 Zeilen mit je 64 Spalten dargestellt.
Es gibt einen Zeichen- und einen Zeilenspeicher. Damit lassen sich Zeichen bzw. Zeilen innerhalb eines Screens oder auch zwischen zwei Screens bewegen oder kopieren. Dabei wird verhindert, daß versehentlich Text verloren geht, indem Funktionen nicht ausgeführt werden, wenn dadurch Zeichen nach unten oder zur Seite aus dem Bildschirm geschoben würden.

2.4.1 HELP und VIEW

Der Editor unterstützt das 'Shadow-Konzept'. Zu jedem Quelltext-Screen gibt es einen Kommentar-Screen. Dieser erhöht die Lesbarkeit von FORTH-Programmen erheblich, Sie wissen ja, guter FORTH-Stil ist selbstdokumentierend ! Auf den Tastendruck CRTL-F9 stellt der Editor den Kommentar-Screen zur Verfügung. So können Kommentare 'deckungsgleich' zu den Quelltexten angefertigt werden. Dieses shadow Konzept wird auch bei dem Wort help ausgenutzt, das zu einem Wort einen erklärenden Text ausgibt. Dieses Wort wird so eingesetzt:

```
help <name>
```

HELP zeigt natürlich nur dann korrekt eine Erklärung an, wenn ein entsprechender Text auf einem shadow screen vorhanden ist.

Häufig möchte man sich auch die Definition eines Wortes ansehen, um z.B. den Stackkommentar oder die genaue Arbeitsweise nachzulesen. Dafür gibt es das Kommando

```
view <name>
```

Damit wird der Screen - und natürlich auch das File - aufgerufen, auf dem <name> definiert wurde. Dieses Verfahren ersetzt (fast) einen Decompiler, weil es

natürlich sehr viel bequemer ist und Ihnen ja auch sämtliche Quelltexte des Systems zur Verfügung stehen.

Werfen Sie doch bitte mit view u? einen Blick auf das Wort, das Ihnen den Inhalt einer Variablen ausgibt. Benutzen Sie

```
fix <name>
```

(engl. = reparieren), so wird zugleich der Editor zugeschaltet, so daß Sie sofort gezielt Änderungen im Quelltext vornehmen können. Im Editor werden Sie zuerst nach einer Eingabe gefragt "Enter your id:", die Sie einfach mit dem Druck auf die <cr>-Taste beantworten.

Dann befinden Sie sich im Editor, mit dem man die Quelltextblöcke bearbeitet. Der Cursor steht hinter dem gefundenen Wort u? . Nun können Sie mit PgUp oder PgDn in den Screens blättern oder mit <ESC> aus dem Editor zum FORTH zurückkehren.

Natürlich müssen für view , fix und help die entsprechenden Files auf den Laufwerken 'griffbereit' sein, sonst erscheint eine Fehlermeldung. Die VIEW-Funktion steht auch innerhalb des Editors zur Verfügung, man kann dann mit dem Tastendruck CTRL-F das rechts vom Cursor stehende Wort anfordern. Dies ist insbesondere nützlich, wenn man eine Definition aus einem anderen File übernehmen möchte oder nicht mehr sicher ist, wie der Stackkommentar eines Wortes lautet.

2.4.2 Öffnen und Editieren eines Files

Nun aber zum Erstellen und Ändern von Quelltexten für Programme - eine schöne Definition von Editieren. Sie haben sich bestimmt eine Datei test.scr wie oben beschrieben mit

```
makefile test.scr  6 more
```

angelegt. Damit haben Sie ein File namens TEST.SCR mit einer Länge von 6 Blöcken (6144 Byte = 6KB), bestehend aus den Screens Ø bis 5 ; davon sind die Nummern Ø bis 2 für Quelltexte, die anderen für Kommentare bestimmt.

Um in den Editor zu gelangen gibt es drei Möglichkeiten:

```
            <screen#> edit
oder        <scr#> l
```

ruft den Screen mit der Nummer <scr#> auf. Hat man bereits editiert, ruft

```
            v
```

den zuletzt bearbeiteten Screen wieder auf. Dies ist der zuletzt editierte oder aber, und das ist sehr hilfreich, derjenige, der einen Abbruch beim Kompilieren verursacht hat. volksFORTH bricht - wie die meisten modernen Systeme - bei einem Programmfehler während des Kompilierens ab und markiert die Stelle, wo der Fehler auftrat. Dann brauchen Sie nur v einzugeben. Der fehlerhafte Screen wird in den Editor geladen und der Cursor steht hinter dem Wort, das den Abbruch des Kompilierens verursachte.

Als Beispiel soll der Block #1 editiert werden:

```
1 edit
```

Sie werden nach der Eingabe des dreibuchstabigen Kürzels gefragt. Dieses Kürzel wird dann rechts oben in den Screen eingetragen und gibt die programmer's id wieder. Wenn Sie nichts eingeben möchten, drücken Sie bitte nur <cr>. Zum Benutzen dieser Blöcke gibt es noch einige Vereinbarungen, von denen ich hier zwei nennen möchte:

1. wird der Block Nr.Ø nie !! für Programmtexte benutzt - dort finden sich meist Erklärungen und Hinweise zum Programm und zum Autor.
2. In die Zeile Ø eines Blockes wird immer ein Kommentar eingetragen, der mit \ (skip line) eingeleitet wird. Dieser Backslash sorgt dafür, daß die nachfolgende Zeile als Kommentar überlesen wird. In diese Zeile Ø schreibt man oft die Namen der Wörter, die im Block definiert werden.

2.4.3 Tastenbelegung des Editors

Beim Editieren stehen Ihnen folgende Funktionen zur Verfügung:

F1	gibt Hilfestellung für den Editor
ESC	verläßt den Editor mit dem sofortigen Abspeichern der Änderungen.
CTRL-U	(undo) macht alle Änderungen rückgängig, die noch nicht auf Disk zurückgeschrieben wurden.
CTRL-E	verläßt den Editor ohne sofortiges Abspeichern.
CTRL-F	(fix) sucht das Wort rechts vom Cursor, ohne den Editor zu verlassen.
CTRL-L	(showload) lädt den Screen ab Cursorposition.
CTRL-N	fügt an Cursorposition eine neue Zeile ein.
CTRL-PgDn	splittet innerhalb einer Zeile diese Zeile.
CTRL-S	(Scr#) legt die Nummer des gerade editierten Screens auf dem Stack ab; z.B. für ein folgendes load oder plist.
CTRL-Y	löscht die Zeile an Cursorposition
CTRL-PgUp	fügt innerhalb einer Zeile den rechten Teil der unteren Zeile an die obere Zeile (join).
TAB	bewegt den Cursor einen großen TABulator vor.
SHIFT TAB	bewegt den Cursor einen kleinen TABulator zurück.
F2	(suchen/ersetzen) erwartet eine Zeichenkette, die gesucht werden soll und eine Zeichenkette, die statt dessen eingefügt werden soll. Wird eine Übereinstimmung gefunden, kann man mit R (replace) die gefundene Zeichenkette ersetzen, mit <cr> den Suchvorgang abbrechen oder mit jeder anderen Taste die nächste Übereinstimmung suchen.

Auf diese Weise kann man die Quelltexte auch nach einer Zeichenkette durchsuchen lassen. Als Ersatz-String wird dann <CR> eingegeben.

F3 bringt eine Zeile in den Zeilenpuffer und löscht sie im Screen.

F5 bringt die Kopie einer Zeile in den Zeilenpuffer.

F7 fügt die Zeile aus dem Zeilenpuffer in den Screen ein.

F4 wie F3, nur für ein einzelnes Zeichen.

F6 wie F5, jedoch für ein einzelnes Zeichen.

F8 entspricht F7, bezogen auf ein Zeichen.

F9 vertauscht die aktuelle Datei (isfile) mit der Hintergrunddatei (fromfile). Erneutes F9-Drücken vertauscht erneut und stellt damit den alten Zustand wieder her. Diese Funktion ist dann sinnvoll, wenn Sie eine Datei bearbeiten, sich zwischendurch aber mit CRTL-F ein Wort anzeigen lassen. Dann stellt F9 (=fswap) die alte Dateiverteilung wieder her, die sich durch das fix geändert hat.

SHIFT F9 schaltet auf die Kommentartexte (shadowscreens) um und beim nächsten Drücken wieder zurück.

F1Ø legt den aktuellen Screen kurz beiseite, um ihn dann mit einem Druck auf F9 wieder bearbeiten zu können. Sollte Ihnen das volksFORTH eine der Kopierfunktionen copy oder convey mit der Meldung TARGET BLOCK NOT EMPTY verweigern, weil isfile und fromfile unterschiedlich sind, so sorgt F1Ø wieder für klare Verhältnisse.

Noch ein Hinweis zum Editor:

Der Editor unterstützt nur das Kopieren von Zeilen. Man kann auf diese Art auch Screens kopieren, aber beim gelegentlich erforderlichen Einfügen von Screens in der Mitte eines Files ist das etwas mühselig. Zum Kopieren ganzer Screens innerhalb eines Files oder von einem File in ein anderes werden im volksFORTH83 die Worte COPY und CONVEY verwendet.

2.4.4 Beispiel: CLS

Für Ihr erstes Programm tragen Sie nun bitte den folgenden Quelltext in Screen 1 ein:

```
\ CLS löscht den gesamten Bildschirm
: cls   ( -- )  full page ;
```

Nun drücken Sie SHIFT-F9 und tragen einen Kommentar in diesen Screen ein, wobei die Erklärung zu CLS in der gleichen Zeile, wie die Definition des Quelltextes, z.B.:

```
\\ CLS

CLS  löscht den gesamten Bildschirm, indem es auf die Worte
     full (Bildschirmfenster auf volle Größe) und PAGE (Bild
     schirmfenster löschen)  zurückgreift.
```

Ein nochmaliges SHIFT-F9 bringt Sie wieder zum Quelltext (source code) zurück.. Damit bekommen Sie diesen erklärenden Text nach dem Kompilieren mit help cls angezeigt.

Wenn Sie einen Block vollgeschrieben haben, blättern Sie nur mit PgUp vor oder mit PgDn zurück zum nächsten Block. Sind Sie mit Ihrem Programm zufrieden, so drücken Sie ESC ; dann wird der geänderte Screen sofort abgespeichert.

Danach werden Sie wieder etwas bemerken: Der Bildschirm arbeitet nicht mehr korrekt, es bleiben oben Zeilen stehen, die nicht scrollen.
Das ist richtig, damit der zuletzt bearbeitete Quelltext nicht nach oben wegscrollt. Um nicht weitere zwei Zeilen des Bildschirms zu verlieren, hat das Fenster, in dem Sie gerade arbeiten, keinen Rahmen. Wie man mit Rahmen und Fenstern arbeitet, wird im Editor vorgeführt, der wie alle anderen Systemteile im Quelltext vorliegt.

Um sich in Zukunft schnell aus der mißlichen Lage mit der reduzierten Bildschirmgröße zu befreien, benutzen Sie Ihr erstes selbstgeschriebenes Programm cls . Denn die Eingabe von full stellt Ihnen wieder die gesamten Bildschirmfläche zur Verfügung, wogegen page nur das aktuelle Fenster löscht.
In volks4TH wird das Kompilieren - wie in den meisten FORTH-Systemen - über <scr#> load durchgeführt:

```
1 load
```

kompiliert Ihr Wort CLS mit für 'C'- oder Pascal-Programmierer unvorstellbarer Geschwindigkeit ins FORTH. Damit steht Ihr Mini-Programm jetzt zur Ausführung bereit. Zugleich erhalten Sie die Meldung, daß ein Wort namens CLS bereits besteht: "CLS exists" . Dies hat nur die Konsequenz, daß nach einer Redefinition das alte Wort gleichen Namens nicht mehr zugegriffen werden kann. Eine Möglichkeit, diese Namensgleichheit mit einem bereits existierenden Wort zu vermeiden, wäre der Einsatz eines Vokabulares.
Geben Sie einmal words ein, dann werden Sie feststellen, daß Ihr neues Wort CLS ganz oben im Dictionary steht (drücken Sie die <ESC>-Taste, um die Ausgabe von words abzubrechen oder irgendeine andere Taste, um sie anzuhalten).
Um das Ergebnis Ihres ersten Programmierversuchs zu überprüfen, geben Sie nun ein:

```
cls
```

Und siehe da, der gesamte Bildschirm wird dunkel. Schöner wäre es allerdings, wenn Ihr Programm seine Arbeit mit einer Meldung beenden würde. Um dies zu ändern, werfen Sie bitte erst einmal das alte Wort CLS weg:

```
forget cls
```

Bitte kontrollieren Sie mit words , ob CLS wirklich vergessen wurde. Rufen Sie dann erneut den Editor mit v auf. Nun benutzen Sie das Wort für den Beginn einer Zeichenkette ." , das Wort für ihr Ende " und das Wort für einen Zeilenvorschub cr . Ihr Screen 1 sieht dann so aus :

```
\ CLS löscht den Bildschirm mit Meldung
```

```
: cls   ( -- )
   full page
   ." Bildschirm ordnungsgemäß gelöscht!" cr ;
```

Nach dem ." muß ein Leerzeichen stehen. Denn FORTH benutzt standardmäßig das Leerzeichen als Trennzeichen zwischen einzelnen Worten, so daß dies Leerzeichen nicht mit zur Zeichenkette (string) zählt. Dann verlassen Sie den Editor und kompilieren Sie Ihr Programm wie gehabt und starten es. Die Änderung erweist sich als erfolgreich, und Sie haben gelernt, wie einfach in FORTH das Schreiben, Austesten und Ändern von Programmteilen ist.
Eine große Hilfe sowohl für den Programmierer als auch für den späteren Benutzer sind Informationen darüber, was gerade geladen wird und was schon kompiliert wurde. Fügt man

```
cr  .( Funktion installiert )
```

ein, so werden während des Ladens diese Meldungen ausgegeben. Das Wort .(leitet einen Kommentar im Interpreter ein, die schließende Klammer) beendet ihn und cr ist natürlich für den Zeilenvorschub, das carriage return, verantwortlich.

2.4.5 Compilieren im Editor - Showload

Eine Besonderheit von volksFORTH ist, daß selbst im Editor Funktionen des Interpreters/Compilers zur Verfügung stehen. Dieses Interpretieren und Kompilieren im Editor nennt sich Showload.

CTRL-F	(fix) sucht und zeigt das Wort rechts vom Cursor, ohne den Editor zu verlassen.
CTRL-L	(showload) lädt den Screen ab Cursorposition.

Um zu sehen, was diese Showload-Funktion leistet, geben Sie nun bitte folgenden Screen ein:

```
\ Ein Test für das showload
  : inc  1+ ;
  : dec  1- ;
\\
  15 inc .
  15 inc dec .
  15 15 +  2 spaces .
```

Dieser Screen soll jetzt im Editor kompiliert und interpretiert werden !!
Dazu setzen Sie bitte den Cursor durch die Taste CTRL-home in die erste Zeile auf das Zeichen \ (skip line). Drücken Sie nun CTRL-L zum Laden des Screens. volksFORTH kompiliert nun bis zu der Zeile, die mit \\ (skip screen) beginnt. Die Wörter inc und dec sind dem System jetzt bekannt und können benutzt werden.
Anschließend bewegen Sie den Cursor hinter das \\ und drücken CTRL-L zum weiteren Interpretieren im Editor. Sofort sehen Sie die Ausgaben an der entsprechenden Stelle im Editor erscheinen. Dabei bleibt der Inhalt des Screens selbst-

verständlich unversehrt - verlassen Sie den Editor mit ESC und sehen Sie sich den Inhalt den Screens mit <scr#> list an; Sie sehen nur die Anweisungen, aber nicht mehr die Ausgaben vor sich.
Eine typische Anwendung dieses showload wäre das Neukompilieren eines Wortes nach einer kleinen Änderung oder das interaktive Hinzufügen von Wörtern, die man gerade mal braucht.

2.5 Erstellen einer Applikation

Sie wollen Ihr 'Programm' nun als eigenständige Anwendung abspeichern. Dazu erweitern Sie es zunächst ein klein wenig (Editor mit v aufrufen.) FÜgen Sie nun noch folgende Definition in einer neuen Zeile hinzu:

```
: runalone    cls bye  ;
```

RUNALONE führt zuerst CLS aus und kehrt dann zum Betriebssystem zurück. Kompilieren Sie nun erneut, wobei Sie die Meldung erhalten "CLS exists". Sie führen RUNALONE aber nicht aus, sonst würden Sie FORTH ja verlassen (bye).
Das Problem besteht vielmehr darin, das System so abzuspeichern, daß es gleich nach dem Laden RUNALONE ausführt und sonst gar nichts. volksFORTH83 ist an zwei Stellen für solche Zwecke vorbereitet. In den Worten COLD und RESTART befinden sich zwei 'deferred words' namens 'COLD bzw. 'RESTART , die im Normalfall nichts tun, vom Anwender aber nachträglich verändert werden können.
Sie benutzen hier 'COLD, um auch schon die Startmeldung zu unterbinden. Geben Sie also ein

```
' runalone Is 'cold
```

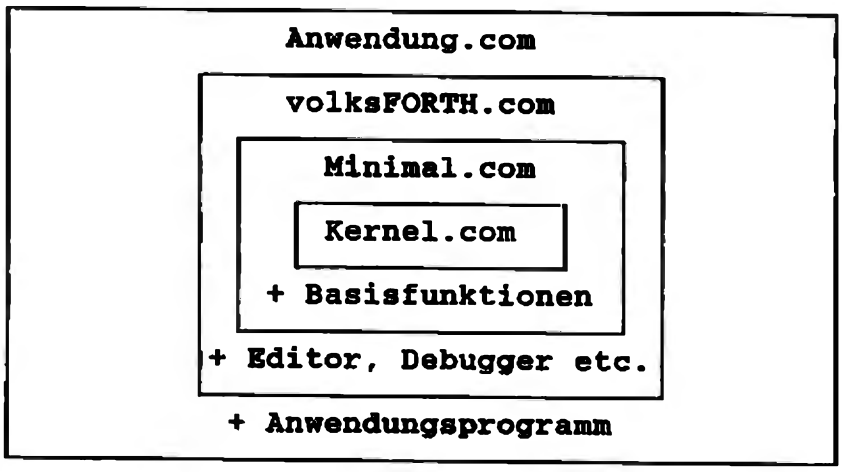
und speichern Sie das Ganze mit

```
savesystem cls.com
```

auf Disk zurück. Sie haben Ihre erste Applikation erstellt, die Sie von MSDOS-Ebene aus mit CLS aufrufen können !

Etwas enttäuschend ist es aber schon. Das angeblich so kompakte FORTH benötigt über 2ØKByte, um eine so lächerliche Funktion auszuführen ?? Da stimmt doch etwas nicht. Natürlich, es wurden ja eine Reihe von Systemteilen mit abgespeichert, vom Fileinterface über den Assembler, den Editor usw., die für unser Programm überhaupt nicht benötigt werden.

Um dieses und ähnliche Probleme zu lösen, gibt es das File KERNEL.COM. Dieses Programm enthält nur den Systemkern und das Fileinterface und entspricht damit der Laufzeit-Bilbliothek (runtime library) anderer Sprachen.



Laden Sie also KERNEL.COM und kompilieren Sie Ihre Applikation mit

```
include multi.vid
include test.scr
```

Das vorherige Laden von MULTI.VID ist nötig, weil FORTH-Systeme selten über Linker verfügen; wird MULTI.VID nicht vorher geladen, so ist dem FORTH-System das Wort full unbekannt. Dann wie gehabt RUNALONE in 'COLD eintragen und das System auf Disk zurückspeichern. Von der MS-DOS Ebene aus läßt sich diese Programmerstellung mit

```
kernel include mulit.vid include test.scr
```

durchführen, wobei diese beiden Zeilen

```
' runalone Is 'cold
savesystem dark.com
```

die letzten Anweisungen auf Ihrem Screen sind. Damit wird der FORTH-Interpreter angewiesen, die fertige Anwendung DARK.com auf Disk zu speichern.
Sie haben jetzt eine verhältnismäßig kompakte Version vorliegen. Natürlich ließe sich auch diese noch erheblich kürzen, aber dafür bräuchten Sie einen Target-Compiler, mit dem Sie nur noch die wirklich benötigten Systemteile selektiv aus dem Quelltext zusammenstellen könnten. Mit der beschriebenen Methode lassen sich aber auch größere Programme kompilieren und als Stand-alone-Applikationen abspeichern.

2.6 Das Erstellen eines eigenen FORTH-Systems

Das File VOLKS4TH.COM ist als Arbeitsversion gedacht.

Es enthält alle wichtigen Systemteile wie Editor, Printer-Interface, Tools, Decompiler, Tracer usw. Sollte Ihnen die Zusammenstellung nicht gefallen, können Sie sich jederzeit ein Ihren speziellen Wünschen angepaßtes System zusammenstellen.

Schlüssel dazu ist der Loadscreen des Files VOLKS4TH.SYS . Sie können dort Systemteile, die Sie nicht benötigen, mit dem Backslash \ wegkommentieren oder die entsprechenden Zeilen ganz löschen. Ebenso können Sie natürlich dem Loadscreen eigene Files hinzufügen. Entspricht der Loadscreen Ihren Wünschen, speichern Sie ihn mit <ESC> zurück, und verlassen Sie das System mit BYE. Laden Sie nun das File KERNEL.COM . Geben Sie dann ein:

```
include volks4TH.sys
```

Ist das System fertig kompiliert, schreibt die Anweisung

```
savesystem volks4TH.com
```

des Load-Screens eine neue Version von volks4TH.com auf Disk. Damit wird Ihr altes File überschrieben (Sicherheitskopie !!!), sodaß Sie beim nächsten Laden von volks4TH.com Ihr eigenes System erhalten.

Natürlich können Sie 'Ihr' System auch unter einem anderen Namen abspeichern. Ebenso können Sie Systemvoreinstellungen ändern. Unsere Arbeitsversion arbeitet - voreingestellt - neuerdings im dezimalen Zahlensystem. Natürlich können Sie mit hex auf Hexadezimalsystem umstellen; wir halten das für sehr viel sinnvoller, weil vor allem Speicheradressen im Dezimalsystem kaum etwas aussagen (oder wissen Sie, ob Speicherstelle 978584 im Bildschirmspeicher liegt oder nicht ?). Wollen Sie bereits unmittelbar nach dem Laden im Hexadezimalsystem arbeiten, können Sie sich dies mit SAVESYSTEM abspeichern, indem Sie von der FORTH-Kommandozeile aus savesystem volks4TH.com eingeben.

Im Übrigen empfehlen wir bei allen Zahlen über 9 dringend die Benutzung der sogenannten Präfixe:

$ für Hexadezimal-,
& für Dezimal- und
% für Binärzahlen.

Man vermeidet so, daß irgendwelche Files nicht - oder noch schlimmer, falsch - kompiliert werden, weil man gerade im anderen Zahlensystem ist. Außerdem ist es möglich, hexadezimale und dezimale Zahlen beliebig zu kombinieren, je nachdem, was gerade sinnvoller ist. In den Quelltexten finden Sie genug entsprechende Beispiele.

Besonders schnell und komfortabel arbeitet volksFORTH natürlich, wenn alle Teile des Systems auf einer Festplatte abgelegt sind. Sie sollten dafür ein eigenes Directory einrichten und PATH und DIR entsprechend einstellen.
Auch die Arbeit mit einer RAM-Disk ist prinzipiell möglich, allerdings nicht sehr zu empfehlen. FORTH ist sehr maschinennah und Systemabstürze daher vor allem zu Anfang nicht so ganz auszuschließen.
Das Ausdrucken der Quelltexte des Systems ist sicher sinnvoll, um Beispiele für den Umgang mit volksFORTH83 zu sehen. So stellen z.B. der Screen-Editor und der Kommando-Editor vollständige Anwendungen dar, die im Quelltext vorliegen.
Welche Files sich im Einzelnen auf Ihren Disketten befinden und ob sie Kommen-

tarscreens enthalten, steht im File README.DOC. Zunächst müssen Sie das Printer-Interface hinzuladen, falls es nicht schon vorhanden ist. Reagiert Ihr volksFORTH83 auf die Eingabe von PRINTER mit einem ? , so ist das Printerinterface nicht vorhanden.

2.7 Ausdrucken von Screens

Sollte in Ihrem System kein Druckerinterface vorhanden sein, so laden Sie es von der FORTH-Kommandozeile aus mit

```
include <Druckername>.prn
```

nach. Sollte Ihr Drucker in der Liste nicht erscheinen, so benutzen Sie statt dessen graphic.prn oder epson.prn . Die meisten Drucker können als IBM-Graphic-Printer oder als EPSON FX/LX-Drucker arbeiten.
Im Printer-Interface sind einige Worte zur Ausgabe eines formatierten Listings enthalten. PTHRU druckt einen Bereich von Screens, jeweils 6 Screens auf einer DIN A4 Seite in komprimierter Schrift. Ganz ähnlich arbeitet das Wort DOCUMENT , jedoch wird bei diesem Wort neben einen Quelltextscreen der zugehörige Shadowscreen gedruckt. LISTING druckt ein ganzes File so aus, man erhält so ein übersichtliches Listing eines Files mit ausführlichen Kommentaren.

Glossar

pthru (von bis --)
druckt die angegebenen Blöcke immer zu sechst auf einer Seite aus.

document (von bis --)
arbeitet wie pthru , druckt aber jeweils drei Quelltextblöcke und drei Kommentarblöcke auf einer Seite aus.

listing (--)
erstellt ein Listing der gesamten Datei, indem jeweils drei Quelltext-Blöcke und drei Kommentar-Blöcke auf einer Seite ausgedruckt werden.

plist (scr# --)
druckt einen angegebenen Block auf dem Drucker aus.

scr (-- addr)
ist eine Variable, die die Nummer des gerade editierten Screens enthält. Vergl. r# , list, (error

r# (-- addr)

ist eine Variable, die den Abstand des gerade editierten Zeichens von Anfang des gerade editierten Screens enthält.

2.8 Druckeranpassung

Die Druckeranpassung das Arbeitssystems wird im File VOLKS4TH.SYS durch die Zeile

```
include <printer>.prn
```

vorgenommen. In dieser Anpassung sind - zusätzlich zu den reinen Ausgaberoutinen - eine Reihe nützlicher Worte enthalten, mit denen die Druckersteuerung sehr komfortabel vorgenommen werden kann.
Im Arbeitssystem ist das Printerinterface bereits enthalten. Müssen Sie Änderungen vornehmen, können Sie mit dem Editor den Loadscreen von VOLKS4TH.SYS ändern und sich ein neues Arbeitssystem zusammenstellen mit:

```
kernel include volks4TH.sys
```

Sie können natürlich auch den Loadscreen in seiner jetzigen Fassung benutzen und das Printer-Interface jedesmal 'von Hand' mit

```
include <printer>.prn
```

nachladen.

Leider sind im Moment im volksFORTH noch deutsche und englische Fehlermeldungen gemischt und die help Funktion zeigt Ihnen einen erklärenden Text nur, wenn dieser vorhanden ist.

Die wichtigsten Befehle noch einmal im Überblick:

status	steuert die Status-Zeile
words	zeigt die gerade verfügbaren Befehle an
files	zeigt die angemeldeten Dateien
path	verändert oder nennt den Datei-Suchpfad
order	listet die Suchreihenfolge der Befehlsgruppen auf
vocs	nennt alle verfügbaren Befehlsgruppen
view	zeigt und
fix	editiert den Quelltext eines bestimmten Wortes
help	zeigt - wenn vorhanden - den Kommentartext eines Wort
full	schaltet das Bildschirmfenster auf die volle Größe
page	löscht das aktuelle Fenster
index	- nicht resident - zeigt den Inhalt einer Block-Datei
list	zeigt den Inhalt eines Screens.

include lädt eine ganze Befehlsgruppe oder einen Programteil

Nun sollten Sie bereits zu einem begleitenden Buch zu greifen. Denn volksFORTH ist ein komplexes multitaskingfähiges Entwicklungssystem, das man nicht von heute auf morgen beherrscht.

[1] Leo Brodie — Programmieren in FORTH (Hanser Verlag)

[2] Leo Brodie — In FORTH denken (Hanser Verlag)

[3] R. Zech — FORTH 83 (Franzis Verlag)

[4] H.-W. Beilstein — Wie man in FORTH programmiert (Chip Wissen)

3. Arithmetik

3.1 Stacknotation

Im folgenden werden hauptsächlich Worte in ihrer Einzelfunktion beschrieben. In dieser Form der Beschreibung, die Sie bereits kennengelernt haben, wird die Wirkung eines Wortes auf den Stack in Klammern angegeben und zwar in folgender Form:

(vorher -- nachher)

vorher : Werte auf dem Stack vor Ausführung des Wortes
nachher : Werte auf dem Stack nach Ausführung des Wortes

In dieser Notation wird das oberste Element des Stacks (tos) immer ganz rechts geschrieben. Sofern nicht anders angegeben, beziehen sich alle Stacknotationen auf die spätere Ausführung des Wortes. Bei immediate Worten wird auch die Auswirkung des Wortes auf den Stack während der Kompilierung angegeben. Worte werden ferner durch folgende Symbole gekennzeichnet:

C	Dieses Wort kann nur während der Kompilation einer :-Definition benutzt werden.
I	Dieses Wort ist ein immediate Wort, das auch im kompilierenden Zustand ausgeführt wird.
83	Dieses Wort wird im 83-Standard definiert und muß auf allen Standardsystemen äquivalent funktionieren.
U	Kennzeichnet eine Uservariable.

Weicht die Aussprache eines Wortes von der natürlichen englischen Aussprache ab, so wird sie in Anführungszeichen angegeben. Gelegentlich folgt auch eine deutsche Übersetzung.
Die Namen der Stackparameter folgen, sofern nicht suggestive Bezeichnungen gewählt wurden, dem nachstehendem Schema. Die Bezeichnungen können mit einer nachfolgenden Ziffer versehen sein.

Stack-notation	Zahlentyp	Wertebereich in Dezimal	minimale Feldbreite
flag	logischer Wert	Ø=falsch, sonst=true	16 Bit
true (tf)	logischer Wert	-1 (als Ergebnis)	16 Bit
false (ff)	logischer Wert	Ø	16 Bit
b	Bit	Ø..1	1 Bit
char	Zeichen	Ø..127 (Ø..256)	7(8Bit)

8b	8 beliebige Bits	nicht anwendbar	8
16b	16 beliebige Bits	nicht anwendbar	16
n	Zahl,bewertete Bits	-32768..32767	16
+n	positive Zahl	Ø..32767	16
u	vorzeichenlose Zahl	Ø..65535	16
w	Zahl, n oder u	-32768..65535	16
addr	Adresse, wie u	Ø..65535	16
32b	32 beliebige Bits	nicht anwendbar	32
d	doppelt genaue Zahl	-2,147,483,648... 2,147,483,647	32
+d	pos. doppelte Zahl	Ø..2,147,483,647	32
ud	vorzeichenlose doppelt genaue Zahl	Ø..4,294,967,295	32
sys	systemabhängige Werte	nicht anwendbar	nicht anwendbar

3.2 Arithmetische Funktionen

```
-1          ( -- -1 )
Ø           ( -- Ø )
1           ( -- 1 )
2           ( -- 2 )
3           ( -- 3 )
4           ( -- 4 )
```

Oft benutzte Zahlenwerte wurden zu Konstanten gemacht. Definiert in der Form :

```
n Constant n
```

Dadurch wird Speicherplatz eingespart und die Ausführungszeit verkürzt.

1+ (w1 -- w2) 83 "one-plus"

w2 ist das Ergebnis von Eins plus w1. Die Operation 1 + wirkt genauso.

1- (w1 -- w2) 83 "one-minus"

w2 ist das Ergebnis von w1 minus Eins. Die Operation 1 - wirkt genauso.

2+ (w1 -- w2) 83 "two-plus"

w2 ist das Ergebnis von w1 plus Zwei. Die Operation 2 + wirkt genauso.

2- (w1 -- w2) 83 "two-minus"

w2 ist das Ergebnis von w1 minus Zwei. Die Operation 2 - wirkt genauso.

2* (w1 -- w2) "two-times"
w1 wird um ein Bit nach links geschoben und das ergibt w2. In das niederwertigste Bit wird eine Null geschrieben. Die Operation 2 * wirkt genauso.

2/ (n1 -- n2) 83 "two-divide"
n1 wird um ein Bit nach rechts verschoben und das ergibt n2. Das Vorzeichen wird berücksichtigt und bleibt unverändert. Die Operation 2 / wirkt genauso.

3+ (w1 -- w2) "three-plus"
w2 ist das Ergebnis von w1 plus Drei. Die Operation 3 + wirkt genauso.

abs (n -- u) 83 "absolute"
u ist der Betrag von n. Wenn n gleich -32768 ist, hat u denselben Wert wie n. Vergleiche auch Zweierkomplement.

not (w1 -- w2) 83
Jedes Bit von w1 wird einzeln invertiert und das ergibt w2.

negate (n1 -- n2) 83
n2 hat den gleichen Betrag, aber das umgekehrte Vorzeichen von n1. n2 ist gleich der Differenz von Null minus n1.

even (u1 -- u2)
ist im 8086-FORTH ein noop-Befehl ohne Funktion.

max (n1 n2 -- n3) 83 "maximum"
n3 ist die Größere der beiden Werte n1 und n2. Benutzt die Operation > . Die größte Zahl für n1 oder n2 ist 32767.

min (n1 n2 -- n3) 83 "minimum"
n3 ist die Kleinere der beiden Werte n1 und n2. Benutzt die Operation < . Die kleinste Zahl für n1 oder n2 ist -32768.

+ (w1 w2 -- w3) 83 "plus"
w1 und w2 addiert ergibt w3.

- (w1 w2 -- w3) 83 "minus"
w2 von w1 subtrahiert ergibt w3.

***** (w1 w2 -- w3) 83 "times"

Der Wert w1 wird mit w2 multipliziert. w3 sind die niederwertigen 16 Bits des Produktes. Ein Überlauf wird nicht angezeigt.

/ (n1 n2 -- n3) 83 "divide"

n3 ist der Quotient aus der Division von n1 durch den Divisor n2. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768...32767) liegt.

mod (n1 n2 -- n3) 83 "mod"

n3 ist der Rest der Division von n1 durch den Divisor n2. n3 hat daßelbe Vorzeichen wie n2 oder ist Null. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768..32767) liegt.

/mod (n1 n2 -- n3 n4) 83 "divide-mod"

n3 ist der Rest und n4 der Quotient aus der Division von n1 durch den Divisor n2. n3 hat dasselbe Vorzeichen wie n2 oder ist Null. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768..32767) liegt.

***/** (n1 n2 n3 -- n4) 83 "times-divide"

Zuerst wird n1 mit n2 multipliziert und ein 32-bit Zwischenergebnis erzeugt. n4 ist der Quotient aus dem 32-bit Zwischenergebnis und dem Divisior n3. Das Produkt von n1 mal n2 wird als 32-bit Zwischenergebnis dargestellt, um eine größere Genauigkeit gegenüber dem sonst gleichwertigen Ausdruck n1 n2 * n3 / zu erhalten. Eine Fehlerbedingung besteht, wenn der Divisor Null ist, oder der Quotient außerhalb des Intervalls (-32768.. 32767) liegt.

***/mod** (n1 n2 n3 -- n4 n5) 83 "times-divide-mod"

Zuerst wird n1 mit n2 multipliziert und ein 32-bit Zwischenergebnis erzeugt. n4 ist der Rest und n5 der Quotient aus dem 32-bit-Zwischenergebnis und dem Divisor n3. n4 hat das gleiche Vorzeichen wie n3 oder ist Null. Das Produkt von n1 mal n2 wird als 32-bit Zwischenergebnis dargestellt, um eine größere Genauigkeit gegenüber dem sonst gleichwertigen Ausdruck n1 n2 * n3 /mod zu erhalten. Eine Fehlerbedingung besteht, falls der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768...32767) liegt.

u/mod (u1 u2 -- u3 u4) "u-divide-mod"

u3 ist der Rest und u4 der Quotient aus der Division von u1 durch den Divisor u2. Die Zahlen u sind vorzeichenlose 16-Bit Werte (unsigned integer). Eine Fehlerbedingung besteht, wenn der Divisor Null ist.

umax (u1 u2 -- u3) "u-maximum"
u3 ist der Größere der beiden Werte u1 und u2. Benutzt die U> Operation. Die größte Zahl für u1 oder u2 ist 65535.

umin (u1 u2 -- u3) "u-minimum"
u3 ist der Kleinere der beiden Werte u1 und u2. Benutzt die U< Operation. Die kleinste Zahl für u1 oder u2 ist Null.

3.3 Logik und Vergleiche

true (-- -1)
hinterläßt -1 als Zeichen für logisch wahr auf dem Stack.

false (-- Ø)
Hinterläßt Null als Zeichen für logisch-falsch auf dem Stack.

Ø= (w -- flag) 83 "zero-equals"
Wenn w gleich Null ist, ist flag wahr.

Ø<> (n -- flag)
Wenn n verschieden von Null ist, ist flag wahr.

Ø< (n -- flag) 83 "zero-less"
Wenn n kleiner als Null (negativ) ist, ist flag wahr. Dies ist immer dann der Fall, wenn das höchstwertige Bit von n gesetzt ist. Deswegen kann dieser Operator zum Testen dieses Bits benutzt werden.

Ø> (n -- flag) 83 "zero-greater"
Wenn n größer als Null ist, ist flag wahr.

= (w1 w2 -- flag) 83 "equals"
Wenn w1 gleich w2 ist, ist flag wahr.

< (n1 n2 -- flag) 83 "less-than"
Wenn n1 kleiner als n2 ist, ist flag wahr. z.B. -32768 32767 < ist wahr. -32768 Ø < ist wahr.

> (n1 n2 -- flag) 83 "greater-than"
Wenn n1 größer als n2 ist, ist flag wahr.z.B. -32768 32767 > ist falsch. -32768 Ø > ist falsch.

u< (u1 u2 -- flag) 83 "u-less-than"
Wenn u1 kleiner als u2 ist, ist flag wahr. Die Zahlen u sind vorzeichenlose 16-Bit Werte. Wenn Adressen verglichen werden sollen, muß U< benutzt werden, sonst passieren oberhalb von 32K seltsame Dinge !

u> (u1 u2 -- flag) 83 "u-greater-than"
Wenn u1 größer als u2 ist, ist flag wahr. Ansonsten gilt das gleiche wie für U< .

and (w1 w2 -- w3) 83
w1 wird mit w2 bitweise logisch UND verknüpft und das ergibt w3.

or (w1 w2 -- w3) 83
w1 wird mit w2 logisch ODER verknüpft und das ergibt w3.

xor (w1 w2 -- w3) 83 "x-or"
w1 wird mit w2 bitweise logisch EXKLUSIV ODER verknüpft und das ergibt w3.

uwithin (u u1 u2 -- flag)
Wenn u1 kleiner oder gleich u und u kleiner u2 ist (u1<=u<u2), ist flag wahr. Benutzt die U< Operation.

case? (16b1 16b2 -- 16b1 false ¦ true) "case-question"
Vergleicht die beiden Werte 16b1 und 16b2 miteinander. Sind sie gleich, verbleibt TRUE auf dem Stack. Sind sie verschieden, verbleibt FALSE und der darunterliegende Wert 16b1 auf dem Stack. Wird z.B. in der folgenden Form benutzt :

```
key
Ascii a case? IF ... exit  THEN
Ascii b case? IF ... exit  THEN
drop
```

Entspricht dem Ausdruck over = dup IF nip THEN .

3.4 32Bit-Worte

extend (n -- d)
Der Wert n wird auf den doppelt genauen Wert d vorzeichenrichtig erweitert. Benutze für das in der Literatur oft auftretende s>d

```
' extend Alias s>d
```

dabs (d -- ud) 83 "d-absolut"

ud ist der Betrag von d. Wenn d gleich -2.147.483.648 ist, hat ud den selben Wert wie d.

dnegate (d1 -- d2) 83 "d-negate"

d2 hat den gleichen Betrag aber ein anderes Vorzeichen als d1.

d+ (d1 d2 -- d3) 83 "d-plus"

d1 und d2 addiert ergibt d3.

d- (d1 d2 -- d3) "d-minus"

d2 minus d1 ergibt d3.

d* (d1 d2 -- d3) "d-times"

d1 multipliziert mit d2 ergibt d3.

d= (d1 d2 -- flag) "d-equal"

Wenn d1 gleich d2 ist, ist flag wahr.

d< (d1 d2 -- flag) 83 "d-less-than"

Wenn d1 kleiner als d2 ist, ist flag wahr.

dØ= (d -- flag) 83 "d-zero-equals"

Wenn d gleich Null ist, ist flag wahr.

m* (n1 n2 -- d) "m-times"

Der Wert von n1 wird mit n2 multiplizert und d ist das doppelt genaue Produkt.

um* (u1 u2 -- ud) 83 "u-m-times"

Die Werte u1 und u2 werden mulitpliziert und das ergibt das doppelt genaue Produkt ud. UM* ist die anderen multiplizierenden Worten zugrundeliegende Routine.

m/mod (d n1 -- n2 n3) "m-divide-mod"

n2 ist der Rest und n3 der Quotient aus der Division der doppelt genauen Zahl d durch den Divisor n1. Der Rest n2 hat dasselbe Vorzeichen wie n1 oder ist Null. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (-32768..32767) liegt.

ud/mod (ud1 u1 -- u2 ud2) "u-d-divide-mod"
u2 ist der Rest und ud2 der doppelt genaue Quotient aus der Divison der doppelt genauen Zahl ud1 durch den Divisor u1. Die Zahlen u sind vorzeichenlose 16-Bit Werte (unsigned integer). Eine Fehlerbedingung besteht, wenn der Divisor Null ist.

um/mod (ud u1 -- u2 u3) 83 "u-m-divide-mod"
u2 ist der Rest und u3 der Quotient aus der Division von ud durch den Divisor u1. Die Zahlen u sind vorzeichenlose Zahlen. Eine Fehlerbedingung besteht, wenn der Divisor Null ist oder der Quotient außerhalb des Intervalls (Ø..65535) liegt.

3.5 Stack-Operationen

Herkömmliche Programmiersprachen enthalten mehr oder weniger ausgeprägt das Konzept der PROZEDUREN:
Für bestimmte Programmfunktionen notwendige Operatoren werden in benannten Programmteilen zusammengefaßt, um diese Programmfunktionen an mehreren Stellen innerhalb eines Programmes über ihren Namen aktivieren zu können. Da FORTH ohne jede Einschränkung prozedural ist, macht FORTH auch keinen Unterschied zwischen Prozeduren und Funktionen oder seinen Operatoren. Alles wird als ein WORT bezeichnet.
FORTH als Programmier-SPRACHE besteht also aus Wörtern.
Somit können FORTH-Wörter sein:

1. Datenbereiche
2. Algorithmen (Befehle)
3. Programme

Um Prozeduren sinnvoll benutzen zu können, kennen die meisten Sprachen auch PARAMETER:
Dies sind Daten, die einer Prozedur bei ihrem Aufruf zur Bearbeitung übergeben werden. Daten, die ausschließlich innerhalb einer Prozedur benötigt werden, heißen LOKAL zu dieser Prozedur; im Gegensatz dazu nennt man Daten, die außerhalb von bestimmten Prozeduren zur Verfügung stehen und auf die von allen Prozeduren aus mit allen Operatoren zugegriffen werden kann, GLOBAL.

Die erste Möglichkeit der Parameterübergabe zwischen Prozeduren ist die Vereinbarung von benannten GLOBALEN Variablen. Diese globalen Variablen sind für die gesamte Laufzeit des Programmes statisch existent und können von allen Prozeduren manipuliert werden.

Eine andere Möglichkeit der Parameterübergabe besteht im Einrichten eines Speicherbereiches, in dem während das Aufrufes eines Wortes namentlich benannte Parameter dynamisch verwaltetet werden.
Diesen Mechanismus für benannte lokale Variable stellt Standard-FORTH nicht zur

Verfügung, weil die Organisation dieser lokalen Variablen mit einem Verlust sowohl der lokalen Daten nach der Ausführung des Wortes als auch einem Verlust in der Ausführungsgeschwindigkeit des Wortes verbunden sind. Für das volks4TH wurde in der VD 1/88 eine Implementierung benannter lokaler Variabler vorgestellt.

FORTH benutzt zur gegenseitigen Übergabe von Parametern an Wörter hauptsächlich den STACK, einen bestimmten Speicherbereich, in dem die Wörter ihre Parameter erwarten. Diese Parameter erhalten keine Namen, sondern ihre Interpretation ergibt sich aus der Position innerhalb des Stack-Speicherbereiches. Daraus resultiert die Vielzahl von Operatoren zur Änderung der Stack-Position eines Wertes, für die FORTH berühmt/berüchtigt ist.

Damit deutlich wird, welche und wieviele Parameter ein Wort benötigt, werden allgemein STACK-KOMMENTARE verwendet:

Der öffnenden runden Klammer folgt eine Aufzählung der Parameter. Dabei steht der Parameter, der als oberstes Stack-Element erwartet wird, ganz rechts. Dann folgt ein " -- ", das die Ausführung des Wortes symbolisieren soll. Anschließend wird der Zustand des Stacks nach der Ausführung des Wortes dargestellt, wobei das oberste Stackelement wieder ganz rechts steht. Die schließende runde Klammer beendet den Stack-Kommentar.

Ein Wort **SQRT**, das die Quadratwurzel einer Integerzahl liefert, würde in FORTH so benannt und beschrieben:

```
sqrt  ( number -- sqrt )
```

Wird dieses neue Wort aufgerufen, so werden alle darin enthaltenen Wörter ausgeführt, eventuell bereitgestellte Parameter bearbeitet und daraus resultierende Ergebnisse auf dem Stack übergeben.
Der Aufruf von Prozeduren erfolgt in FORTH implizit durch die Nennung des Namens, ebenso wie auch die Datenübergabe zwischen Wörtern meist implizit erfolgt.

3.5.1 Datenstack-Operationen

drop (16b --) 83
Der Wert 16b wird vom Stack entfernt.

2drop (32b --) 83 "two-drop"
Der Wert 32b wird vom Stack entfernt.

dup (16b -- 16b 16b) 83
Der Wert 16b wird dupliziert.

?dup (16b -- 16b 16b | Ø) 83 "question-dup"
Nur wenn der Wert 16b von Null verschieden ist, wird er verdoppelt.

2dup (32b -- 32b 32b) 83 "two-dup"
Der Wert 32b wird dupliziert.

swap (16b1 16b2 -- 16b2 16b1) 83
Die beiden obersten 16-Bit Werte werden vertauscht.

2swap (32b1 32b2 -- 32b2 32b1) 83 "two-swap"
Die beiden obersten 32-Bit Werte 32b1 und 32b2 werden vertauscht.

nip (16b1 16b2 -- 16b2)
Der Wert 16b1, der unter 16b2 auf dem Stack liegt, wird vom Stack entfernt.

over (16b1 16b2 -- 16b1 16b2 16b1) 83
Der Wert 16b1 wird über 16b2 herüberkopiert.

2over (32b1 32b2 -- 32b1 32b2 31b1) "two-over"
Der Wert 32b1 wird über den Wert 32b2 herüber kopiert.

under (16b1 16b2 -- 16b2 16b1 16b2)
Eine Kopie des obersten Wertes auf dem Stack wird unter dem zweiten Wert eingefügt.

rot (16b1 16b2 16b3 -- 16b2 16b3 16b1) 83
Die drei obersten Werte auf dem Stack werden rotiert, sodaß der unterste zum obersten wird.

-rot (16b1 16b2 16b3 -- 16b3 16b1 16b2) "minus-rot"
Die drei obersten 16b Werte werden rotiert, sodaß der oberste Wert zum Untersten wird. Hebt rot auf.

roll (16bn 16bm..16bØ +n -- 16bm..16bØ 16bn) 83
Das +n-te Glied einer Kette von n Werten wird nach oben auf den Stack gerollt. Dabei wird +n selbst nicht mitgezählt.

-roll (16bn .. 16b1 16bØ +n -- 16bØ 16bn .. 16b1)
Das oberste Glied einer Kette von +n Werten wird an die n-te Position gerollt. Dabei wird +n selbst nicht mitgezählt.
2 -roll wirkt wie -rot , Ø -roll verändert nichts.

pick (16bn..16bØ +n -- 16bn..16bØ 16bn) 83

Der +n-te Wert auf dem Stack wird nach oben auf den Stack kopiert. Dabei wird +n selbst nicht mitgezählt.
Ø pick wirkt wie dup , 1 pick wie over.

.s (--) "dot-s"

Gibt alle Werte, die auf dem Stack liegen aus, ohne den Stack zu verändern. Oft benutzt, um neue Worte auszutesten. Die Ausgabe der Werte erfolgt von links nach rechts, der oberste Stackwert zuerst, so daß der top of stack (tos) ganz links steht!

clearstack (... --)

Löscht den Datenstack. Alle Werte, die sich vorher auf dem Stack befanden, sind verloren.

depth (-- n)

n ist die Anzahl der Werte, die auf dem Stack lagen, bevor DEPTH ausgeführt wurde.

sØ (-- addr) "s-zero"

addr ist die Adresse einer Uservariablen, in der die Startadresse des Stacks steht. Der Ausdruck sØ @ sp! wirkt wie clearstack und leert den Stack.

sp! (addr --) "s-p-store"

Setzt den Stackzeiger (stack pointer) auf die Adresse addr. Der oberste Wert auf dem Stack ist dann der, welcher in der Adresse addr steht.

sp@ (-- addr) "s-p-fetch"

Holt die Adresse addr aus dem Stackzeiger. Der oberste Wert im Stack stand in der Speicherstelle bei addr, bevor sp@ ausgeführt wurde.

3.5.2 Returnstack-Operationen

rdepth (-- n) "r-depth"

n ist die Anzahl der Werte, die auf dem Returnstack liegen.

>r (16b --) C,83 "to-r"

Der Wert 16b wird auf den Returnstack gelegt. Siehe auch R> .

r> (-- 16b) C,83 "r-from"

Der Wert 16b wird vom Returnstack geholt. Vergleiche R>.

r@ (-- 16b) C,83 "r-fetch"
Der Wert 16b ist eine Kopie des obersten Wertes auf dem Returnstack.

rdrop (--) C "r-drop"
Der oberste Wert wird vom Returnstack entfernt. Der Datenstack wird nicht verändert. Entspricht der Operation r> drop .

rØ (-- addr) U "r-zero"
addr ist die Adresse einer Uservariablen, in der die Startadresse des Returnstacks steht.

rp@ (-- addr) "r-p-fetch"
Holt die Adresse addr aus dem Returnstackzeiger. Der oberste Wert im Returnstack steht in der Speicherstelle bei addr.

rp! (addr --) "r-p-store"
Setzt den Returnstackzeiger (return stack pointer) auf die Adresse addr. Der oberste Wert im Returnstack ist nun der, welcher in der Speicherstelle bei addr steht.

push (addr --)
Der Inhalt aus der Adresse addr wird bis zum nächsten EXIT oder ; auf dem Returnstack verwahrt und sodann nach addr zurückgeschrieben. Dies ermöglicht die lokale Verwendung von Variablen innerhalb einer :-Definition. Wird z.B. benutzt in der Form :

```
: hex.  ( n -- )  base push hex . ;
```

Hier wird innerhalb von HEX. in der Zahlenbasis HEX gearbeitet, um eine Zahl auszugeben. Nachdem HEX. ausgeführt worden ist, besteht die gleiche Zahlenbasis wie vorher, durch HEX wird sie also nur innerhalb von HEX. verändert.

4. Kontrollstrukturen

4.1 Programm-Strukturen

Wil Baden, auf den Sie in der englischsprachigen Literatur oft stoßen, hat in seinem Beitrag ESCAPING FORTH folgendes dargelegt:
Es gibt vier Arten von Steueranweisungen :

- die Abfolge von Anweisungen,
- die Auswahl von Programmteilen,
- die Wiederholung von Anweisungen und Programmteilen
- den Abbruch.

Die ersten drei Möglichkeiten sind zwingend notwendig und in den älteren Sprachen wie PASCAL ausschließlich vorhanden. Entsprechend steht im volksFORTH eine Anweisung für die Auswahl von Programmteilen zur Verfügung, wobei die Ausführung vom Resultat eines logischen Ausdrucks abhängig gemacht wird:

```
flag IF <Anweisungen> THEN
flag IF <Anweisungen> ELSE <Anweisungen> THEN
```

Soll dagegen im Programm ein Rücksprung erfolgen, um Anweisungen wiederholt auszuführen, wird bei einer gegebenen Anzahl von Durchläufen diese Anweisung eingesetzt, wobei der aktuelle Index über I und J zur Verfügung steht:

```
<Grenzen> DO / ?DO <Anweisungen> LOOP
<Grenzen> DO / ?DO <Anweisungen> <Schrittweite> +LOOP
```

Wenn eine Wiederholung von Anweisungen ausgeführt werden soll, ohne daß die Anzahl der Durchläufe bekannt ist, so ist eine Indexvariable mitzuführen oder sonstwie zum Resultat eines logischen Ausdrucks zu kommen. Die folgende Konstruktion ermöglicht eine Endlos-Schleife:

```
BEGIN <Anweisungen> REPEAT
```

Die Wiederholungsanweisungen sind insoweit symmetrisch, daß eine Anweisung solange (while) ausgeführt wird, wie ein Ausdruck wahr ist, oder eine Anweisung wiederholt wird, bis (until) ein Ausdruck wahr wird.

```
BEGIN <Anweisungen> flag UNTIL
BEGIN <Anweisungen> flag WHILE <Anweisungen> REPEAT
```

Beide Möglichkeiten lassen sich in volksFORTH auch kombinieren, wobei auch mehrere (multiple) **WHILE** in einer Steueranweisung auftreten dürfen.

```
BEGIN <Anweisungen>  flag WHILE <Anweisungen>  flag UNTIL
```

Nun tritt in Anwendungen häufig der Fall auf, daß eine Steueranweisung verlassen werden soll, weil sich etwas ereignet hat.

Dann ist die vierte Situation, der Abbruch, gegeben. C stellt dafür die Funktionen: break, continue, return und exit zur Verfügung; volksFORTH bietet hier **exit leave endloop quit abort abort"** und **abort(** an.
In FORTH wird **EXIT** dazu benutzt, um die Definition zu verlassen, in der es erscheint; **LEAVE** dagegen verläßt die kleinste umschließende DO...LOOP-Schleife.

Glossar

Ab der Version 3.81.3 verfügt volksFORTH über eine zusätzliche Steueranweisung für den Compiler, die bedingte Kompilierung in der Form:

have ‹word› **not .IF** ‹action1› **.ELSE** ‹action2› **.THEN**

Diese Worte werden außerhalb von Colon-Definitionen eingesetzt und ersetzen das **\needs** früherer Versionen.

have (-- flag)
prüft, ob ein Wort im Wörterbuch in der Suchreihenfolge existiert und hinterläßt ein entsprechendes Flag. In der Literatur und in Quelltexten findet man auch **exists?** als Synonym.

exit (--)
ist ein Synonym für **unnest** .
Dieses wird von ";" (Semicolon) als Abschluß einer Colondefinition compiliert. Ebenso dient **EXIT** als Austritt aus einem Wort, wobei das Programm in der aufrufenden Ebene fortgesetzt wird (return to caller).
Ein **EXIT** ist innerhalb von DO..LOOP-Strukturen nicht ohne weiteres möglich (siehe **endloop**).
Diese Steueranweisung ist nicht umkehrbar, der Sprung in eine hierarchisch niedrigere Ebene ist nicht erlaubt.
Typisch: **flag IF exit THEN**

?exit (flag --) "question-exit"
führt **EXIT** aus, falls das Flag wahr ist. Ist das Flag falsch, so geschieht nichts. Hierbei soll daran erinnert werden, daß jede Zahl ungleich NULL als wahr interpretiert wird.
Entspricht: **true IF exit THEN**

Ø=exit (flag --) "zero-equals-exit"
führt **EXIT** aus, falls das Flag falsch ist. Ist das Flag wahr, so geschieht nichts. Es entspricht **Ø= IF exit THEN** und wird typisch so eingesetzt:

```
key #cr - Ø=exit
```

IF (flag --) 83,I,C
(-- sys) compiling

wird in der folgenden Art benutzt:

flag IF ... ELSE ... THEN

oder: flag IF ... THEN

Ist das Flag wahr, so werden die Worte zwischen **IF** und **ELSE** ausgeführt und die Worte zwischen **ELSE** und **THEN** ignoriert.
Der **ELSE**-Teil ist optional. Ist das Flag falsch, so werden die Worte zwischen **IF** und **ELSE** (bzw. zwischen **IF** und **THEN** , falls **ELSE** nicht vorhanden ist) ignoriert.

.IF ist das **IF** für den Interpretermodus.

THEN (--) 83,I,C
(sys --) compiling

wird in der folgenden Art benutzt:

IF (...ELSE) ... THEN

Hinter **THEN** ist die Programmverzweigung zuende.
Weil viele FORTH-Freunde die "alte" Schreibweise vor der Festlegung des 83er Standards besser und deutlicher finden, sei hier die Definition von **ENDIF** gezeigt:

' THEN Alias ENDIF immediate restrict

.THEN ist das **THEN** für den Interpretermodus.

ELSE (--) 83,I,C
(sys1 -- sys2) compiling

wird in der folgenden Art benutzt:

flag IF ... ELSE ... THEN

ELSE wird unmittelbar nach dem Wahr-Teil, der auf IF folgt, ausgeführt. **ELSE** setzt die Ausführung unmittelbar hinter **THEN** fort.

.ELSE ist das **ELSE** für den Interpretermodus.

DO (w1 w2 --) 83,I,C
(sys --) compiling

beginnt eine Schleife und entspricht somit ?DO , jedoch wird der Schleifenrumpf mindestens einmal durchlaufen. Der Schleifenindex beginnt mit w2, Grenze ist w1 .
Ist w1=w2 , so wird der Schleifenrumpf 65536-mal durchlaufen.

?DO (w1 w2 --) 83,I,C "question-do"
(-- sys) compiling

wird in der folgenden Art benutzt:

?DO ... LOOP bzw. ?DO ... +LOOP

Beginnt eine Schleife. Der Schleifenindex beginnt mit w2, Limit ist w1. . Ist w2=w1, so wird der Schleifenrumpf überhaupt nicht durchlaufen.
Für Details über die Beendigung von Schleifen siehe +LOOP

LOOP (--) 83,I,C
(-- sys) compiling

entspricht +LOOP, jedoch mit einer festen Schrittweite von 1 .

+LOOP (n --) 83,I,C "plus-loop"
(sys --) compiling

Die Schrittweite n wird zum Loopindex addiert. Falls durch die Addition die Grenze zwischen limit-1 und limit überschritten wurde, so wird die Schleife beendet und die Loop-Parameter werden entfernt. Wurde die Schleife nicht beendet, so wird sie hinter dem korrespondierenden DO bzw. ?DO fortgesetzt.

I (-- w) 83,C

wird zwischen DO und LOOP benutzt, um eine Kopie des Schleifenindex auf den Stack zu holen.

J (-- w) 83,C

wird in zwei geschachtelten DO...LOOP-Schleifen zwischen DO .. DO und LOOP .. LOOP benutzt, um eine Kopie des Schleifenindex der äusseren Schleife auf den Stack zu holen.

leave (--) 83,C

beendet die zugehörige Schleife und setzt die Ausführung des Programmes hinter dem nächsten LOOP oder +LOOP fort. Mehr als ein LEAVE pro Schleife ist möglich, ferner kann LEAVE zwischen anderen Kontrollstrukturen auftreten. Der FORTH83-Standard schreibt abweichend vom volksFORTH vor, daß LEAVE ein immediate Wort ist.

endloop (--)

ermöglicht ein EXIT innerhalb einer DO...LOOP. Es wird so eingesetzt:

```
<Grenzen>DO
        <Anweisungen>
        flag IF endloop exit THEN
       LOOP
```

Damit kann exit auch in einer DO...LOOP-Schleife verwendet werden, vorausgesetzt, Sie halten sich an zwei Regeln:

1. Sie dürfen das System nicht mit Returnstack-Manipulationen aus dem Gleichgewicht gebracht haben.
2. Bei geschachtelten DO...LOOP-Schleife muß für jede Schleifenebene ein endloop dem abbrechenden exit vorangehen.

In der Literatur findet man auch UNDO als Synonym für ENDLOOP.

bounds (start count -- limit start)

dient dazu, ein Intervall, das durch Anfangswert und Länge gegeben ist, in ein Intervall umzurechnen, das durch Anfangswert und Endwert+1 beschrieben wird. Beispiel:

```
10 3 bounds DO...LOOP
```

führt dazu, das I die Werte 10 11 12 annimmt.

BEGIN (--) 83,I,C

(sys ---) compiling

wird in der folgenden Art benutzt:

```
BEGIN ( ...flag WHILE ) ... flag UNTIL
```

oder:

```
BEGIN ( ...flag WHILE ) ... REPEAT
```

BEGIN markiert den Anfang einer Schleife. Der ()-Ausdruck ist optional und kann beliebig oft auftreten. Die Schleife wird wiederholt, bis das Flag vor UNTIL wahr oder oder das Flag vor WHILE falsch ist. REPEAT setzt die Schleife immer fort.

Die Schleife BEGIN <Anweisungen> flag UNTIL wird immer mindestens einmal durchlaufen, da die Abbruchbedingung erst nach dem ersten Durchlauf geprüft wird.

Um dazu vollständige Symmetrie zu erreichen, kann im Ausdruck

```
BEGIN <action> flag WHILE.<action> REPEAT
```

der Anweisungsteil vor WHILE entfallen:

```
BEGIN flag WHILE <Anweisungen> REPEAT
```

prüft die Abbruchbedingung vor dem Eintritt in die Schleife.

REPEAT (--) 83,I,C
(-- sys) compiling

wird in der folgenden Form benutzt:

```
BEGIN (.. WHILE) .. REPEAT
```

REPEAT setzt die Ausführung der Schleife unmittelbar hinter **BEGIN** fort. Der ()-Ausdruck ist optional und kann beliebig oft auftreten. Deshalb gibt es kein **AGAIN** , benutzen Sie statt dessen **REPEAT** oder definieren: `' REPEAT Alias AGAIN immediate restrict`

UNTIL (flag --) 83,I,C
(sys --) compiling

wird in der folgenden Art benutzt:

```
BEGIN (... flag WHILE) ... flag UNTIL
```

Markiert das Ende einer Schleife, deren Abbruch durch flag herbeigeführt wird. Ist das Flag vor **UNTIL** wahr, so wird die Schleife beendet, ist es falsch, so wird die Schleife unmittelbar hinter **BEGIN** fortgesetzt.

WHILE (flag --) 83,I,C
(sys1 -- sys2) compiling

wird in der folgenden Art benutzt:

```
BEGIN .. flag WHILE .. REPEAT
```

oder:

```
BEGIN .. flag WHILE .. flag UNTIL
```

Ist das Flag vor **WHILE** wahr, so wird die Ausführung der Schleife bis **UNTIL** oder **REPEAT** fortgesetzt, ist es falsch, so wird die Schleife beendet und das Programm hinter **UNTIL** bzw. **REPEAT** fortgesetzt. Es können mehrere **WHILE** in einer Schleife verwendet werden.

execute (addr --) 83

Das Wort, dessen Kompilationsadresse addr ist, wird ausgeführt.

perform (addr --)

addr ist eine Adresse, unter der sich ein Zeiger auf die Kompilationsadresse eines Wortes befindet. Dieses Wort wird ausgeführt. Entspricht der Sequenz `@ execute` .

case? (16b1 16b2 -- 16b1 false ¦ true) "case-question"

vergleicht die beiden Werte 16b1 und 16b2 miteinander.
Sind sie gleich, verbleibt **TRUE** auf dem Stack. Sind sie verschieden, verbleibt **FALSE** und der darunterliegende Wert 16b1 auf dem Stack.
Wird z.B. in der folgenden Form benutzt :

```
key
    Ascii a case? IF ... exit  THEN
    Ascii b case? IF ... exit  THEN
drop
```

stop? (-- flag) "stop-question"

ist ein komfortables Wort, das es dem Benutzer gestattet, einen Programmablauf anzuhalten oder zu beenden.
Steht vom Eingabegerät ein Zeichen zur Verfügung, so wird es eingelesen. Ist es #ESC oder CTRL-C , so ist flag **TRUE**, sonst wird auf das nächste Zeichen gewartet. Ist dieses jetzt #ESC oder CTRL-C , so wird **STOP?** mit **TRUE** verlassen, sonst mit **FALSE**.
Steht kein Zeichen zur Verfügung, so ist das Flag **FALSE** . **STOP?** prüft also einen Tastendruck auf #ESC oder CTRL-C .

4.2 Worte zur Fehlerbehandlung

Diese arbeiten auch wie Steueranweisungen, wie die Definitionen von **ARGUMENTS** und **IS-DEPTH** zeigen:

```
: is-depth  ( n -- )
   depth 1-  - abort" falsche Parameterzahl!" ;
```

IS-DEPTH überprüft den Stack auf eine gegebene Anzahl Stackelemente (depth) hin.

abort (--) 83,I

leert den Stack, führt **END-TRACE** 'ABORT **STANDARDI/O** und **QUIT** aus.

'abort (--) "tick-abort"

ist ein deferred Wort, das mit **NOOP** vorbesetzt ist. Es wird in **ABORT** ausgeführt, bevor **QUIT** aufgerufen wird.

abort" (flag) 83,I,C "abort-quote"
(--) compiling

wird in der folgenden Form benutzt:

```
flag Abort" ccc"
```

Wird **ABORT"** später ausgeführt, so geschieht nichts, wenn das Flag falsch ist. Ist das Flag wahr, so wird der Stack geleert und der Inhalt von **ERRORHANDLER** ausgeführt. Beachten Sie bitte, daß im Gegensatz zu **ABORT** kein **END-TRACE** ausgeführt wird.

error" (flag) I,C "error-quote"
(--) compiling

Dieses Wort entspricht **ABORT"** , jedoch mit dem Unterschied, daß der Stack nicht geleert wird.

errorhandler (-- addr)

addr ist die Adresse einer Uservariablen, deren Inhalt die Kompilationsadresse eines Wortes ist. Dieses Wort wird ausgeführt, wenn das Flag, das ABORT" bzw. ERROR" verbrauchen, wahr ist. Der Inhalt von ERRORHANDLER ist normalerweise (ERROR.

(error (string --) "paren-error"

Dieses Wort steht normalerweise in der Variablen ERRORHANDLER und wird daher bei ABORT" und ERROR" ausgeführt. string ist dann die Adresse des auf ABORT" bzw. ERROR" folgenden Strings. (ERROR gibt das letzte Wort des Quelltextes gefolgt von dem String auf dem Bildschirm aus. Die Position des letzten Wortes im Quelltext, bei dem der Fehler auftrat, wird in SCR und R# abgelegt.

r# (-- addr) "r-sharp"

addr ist die Adresse einer Variablen, die den Abstand des gerade editierten Zeichens vom Anfang des gerade editierten Screens enthält.
Vergleiche (ERROR und SCR .

scr (-- addr) 83 "s-c-r"

addr ist die Adresse einer Variablen, die die Nummer des gerade editierten Screens enthält.
Vergleiche R# , (ERROR und LIST .

quit (--) "quit"

entleert den Returnstack und schaltet den interpretierenden Zustand ein.

?pairs (n1 n2 --) "question-pairs"

Ist n1 <> n2 , so wird die Fehlermeldung "unstructured" ausgegeben. Dieses Wort wird benutzt, um die korrekte Schachtelung der Kontrollstrukturen zu überprüfen.

4.3 Fallunterscheidung in FORTH

4.3.1 Strukturierung mit IF ELSE THEN / ENDIF

An dieser Stelle soll kurz die vielfältigen Möglichkeiten gezeigt werden, mit denen eine Fallunterscheidung in FORTH getroffen werden kann. Kennzeichnend für eine solche Programmsituation ist, daß von verschiedenen Möglichkeiten des Programmflusses genau eine ausgesucht werden soll.

Ausgehend von einer übersichtlichen Problemstellung, einem Spiel, werden die notwendigen Grundlagendefinitionen und die Entwicklung der oben beschriebenen Kontrollstruktur beschrieben.

Als Beispiel dient ein Spiel mit einfachen Regeln:

Bei diesem Trinkspiel, das nach [1] auch CRAPS genannt wird, geht es darum, einen Vorrat von gefüllten Gläsern unter den Mitspielern mit Hilfe des Würfels zu verteilen und leerzutrinken:

- Bei einer EINS wurde ein Glas aus dem Vorrat in der Tischmitte genommen und vor sich gestellt.
- Bei einer ZWEI oder einer DREI bekam der Nachbar/ die Nachbarin links ein Glas des eigenen Vorrates zugeschoben.
- Bei einer VIER oder einer FÜNF wurde dem Nachbarn/ der Nachbarin rechts ein Glas des eigenen Vorrates vorgesetzt.
- Bei einer SECHS wurden alle Gläser, die der Spieler/ die Spielerin vor sich stehen hatte, leergetrunken.

Zuordnung ist also: 1=nehmen, 2/3=links, 4/5=rechts, 6=trinken und entsprechend der Augenzahl des Würfels soll eine der 6 möglichen Aktionen ausgeführt werden. Das Programm soll sich darauf beschränken, das Ergebnis dieses Würfelns einzulesen und auszuwerten. Daraufhin wird eine Meldung ausgegeben, welche der sechs Handlungen auszuführen ist.

Für ein solches Programm ist eine Zahleneingabe notwendig. Diese wurde hier mit dem Wort **F83-NUMBER?** realisiert:

```
: F83-number?   ( string -- d f )
  number? ?dup
  IF
     0<  IF extend THEN
     true exit
  THEN
  drop  0 0  false  ;

: input#   ( <string> -- n )
  pad c/l 1- >expect
  pad F83-number?  2drop ;
```

Die Definition der Wörter, die sechs oben genannten Aktionen symbolisch ausführen sollen, richtet sich nach den Spielregeln, die für jedes Würfelergebis genau eine Handlung vorschreiben:

```
\ nehmen  trinken  links  rechts  schieben

: nehmen   bright ." ein Glas nehmen"         normal  2 spaces ;
: trinken  bright ." alle Gläser austrinken"  normal  2 spaces ;
: links    bright ." ein Glas nach LINKS"     normal  2 spaces ;
: rechts   bright ." ein Glas nach RECHTS"    normal  2 spaces ;

: schieben  ;
```

SCHIEBEN ist eine Dummyprozedur, ein Füllsel, dessen Notwendigkeit sich erst sehr spät ergibt.
Für den Dialog mit dem Anwender wird definiert:

```
: Anfrage       cr ." Sollen Sie nehmen, trinken oder schieben? "
                cr ." Bitte Ihre Augenzahl und <cr> : "   ;

: Glückwunsch  cr ." Viel Glück beim nächsten Wurf ... " ;
```

Das Wort **AUSWERTUNG** soll entsprechend einem Selektor genau eine von 6 möglichen Prozeduren ausführen. Also wird man prüfen, ob diese oder diese oder ... der Möglichkeiten in Frage kommt. Hinzu kommt noch die Prüfung, ob der übergebene Parameter zwischen (between) 1 und 6 lag.
Die Definition von **BETWEEN** ist volksFORTH-gemäß recht kurz:

```
( Wert Untergrenze Obergrenze -- false  oder )
(                             -- true   wenn Untergrenze≤Wert≤Obergrenze )
: between   1+ uwithin ;

: Auswertung.1   ( Wurfergebnis -- )
   dup 1 =   IF   nehmen  ELSE
             dup 2 = IF links schieben ELSE
             dup 3 = IF links schieben ELSE
                     dup 4 = IF rechts schieben ELSE
                     dup 5 = IF rechts schieben ELSE
                             dup 6 = IF trinken THEN
                             THEN
                             THEN
                     THEN
                     THEN
             THEN
      1 6 between not IF invers ." Betrug!" normal THEN ;
```

Ein Verzicht auf den **ELSE**-Teil führt schon zu einer übersichtlicheren Form:

```
: Auswertung.2  ( Wurfergebnis --)
     dup 1 = IF nehmen          THEN
     dup 2 = IF links  schieben THEN
     dup 3 = IF links  schieben THEN
     dup 4 = IF rechts schieben THEN
     dup 5 = IF rechts schieben THEN
     dup 6 = IF trinken         THEN

     1 6 between not IF inverse ." Betrug!" normal THEN ;
```

Da eine solche Prüfung auf Gleichheit in der Programmierpraxis oft vorkommt, stellt das volks4TH dafür das Wort **case?** zur Verfügung. **case?** vergleicht die obersten beiden Stackwerte miteinander. Bei Ungleichheit bleibt der Testwert (Selektor) erhalten, so daß die Worte DUP und = dadurch ersetzt werden.

```
: Auswertung.3   ( Wurfergebnis -- )

      1 case? IF nehmen          exit THEN
      2 case? IF links  schieben exit THEN
      3 case? IF links  schieben exit THEN
      4 case? IF rechts schieben exit THEN
      5 case? IF rechts schieben exit THEN
```

```
        6 case? IF trinken          exit THEN

        drop    invers ." Betrug!" normal    ;
```

Bei dieser Auswertung wird aus dem Quelltext zu wenig deutlich, daß bei ZWEI und DREI dieselbe Handlung ausgeführt wird, wie auch VIER und FÜNF die gleichen Aktionen zur Folge haben.

=OR prüft deshalb einen Testwert n2 auf Gleichheit mit einer unter einem Flag f1 liegenden Zahl n2. Das Ergebnis dieses Tests wird mit dem bereits vorliegenden Flag OR-verknüpft. Dieses neue Flag f2 und der Testwert n1 werden übergeben:

```
code =or        ( n1 f1 n2 -- n1 f2 )
   A D xchg  D pop
   S W mov
     W ) A cmp
     0= ?[ -1 # D mov ]?
  next
end-code

\ : =or  ( n1 f1 n2 -- n1 f2 )  2 pick = or  ;
```

Dieses Wortes bringt im Quelltext eine deutliche Verbesserung:

```
: Auswertung.4  ( Wurfergebnis --)
    dup
    1 6 between IF
                 dup 1 =       IF  nehmen           THEN
                 dup 2 = 3 =or IF  links schieben   THEN
                 dup 4 = 5 =or IF  rechts schieben  THEN
                 dup 6 =       IF  trinken          THEN
                ELSE
                 invers ." Betrug!" normal
                THEN
    drop   ;
```

Damit wurde ohne eine CASE-Anweisung eine sehr übersichtliche Steuerung des Programm-Flusses geschaffen.
Die Plausibilitätsprüfung, ob die eingegeben Zahl zwischen 1 und 6 lag, ist hier an den Anfang gerückt und wird in einem einzigen ELSE-Zweig abgearbeitet.

4.3.2 Behandlung einer CASE - Situation

4.3.2.1 Strukturelles CASE

Viele Programmiersprachen eine CASE-Anweisung zur Verfügung, die wie in PASCAL mit Hilfe eines Fall-Indices eine Liste von Fall-Konstanten auswertet und eine entsprechende Anweisung ausführt.
Obwohl ein solches CASE-Konstrukt - wie oben gezeigt - nicht notwendig ist, macht es Programme besser lesbar und liegt bei Problemstellungen wie der Auswer-

tung eines gegebenen Index eigentlich näher.
Dies ist in [1] ausführlich diskutiert worden, wobei aber der ältere Eaker-CASE [2] von Dr. Charles Eaker sicherlich der bekanntere ist, der auch in der Literatur und in Quelltexten häufig Erwähnung und Verwendung findet.

Herr H. Schnitter hat diesen Eaker-CASE für das volks4TH implementiert und dabei Veränderungen in der Struktur und Verbesserungen in der Anwendung vorgenommen.

```
\ caselist  initlist  >marklist  >resolvelist

! variable caselist

! : initlist     ( list -- addr )
                 dup @ swap off
  ;

! : >marklist    ( list -- )
                 here over @ , swap !
  ;

! : >resolvelist ( addr list -- )
                 BEGIN dup @
                 WHILE dup dup @ dup @ rot ! >resolve
                 REPEAT !
  ;

\ case  elsecase  endcase

: CASE           caselist initlist 4
; immediate restrict

: ELSECASE       4 ?pairs
                 compile drop 6
; immediate restrict

: ENDCASE        dup 4 =
                 IF   drop compile drop
                 ELSE 6 ?pairs
                 THEN caselist >resolvelist
; immediate restrict

\ of  endof

: OF             4 ?pairs
                 compile over
                 compile =
                 compile ?branch
                 >mark compile drop 5
; immediate restrict

: ENDOF          5 ?pairs
                 compile branch
                 caselist >marklist
                 >resolve 4
; immediate restrict
```

Diese Implementierung des Eaker-CASE stellt eine Verbesserung gegenüber dem Original dar, indem Herr Schnitter die Kontrollstruktur um **ELSECASE** erweitert hat. Selbstverständlich ist die neue Version vollkommen aufwärtskompatibel mit der Original-version.

Verbesserung:

In der Originalversion der CASE-Struktur ist es nicht möglich, zwischen dem letzten **ENDOF** und **ENDCASE** einen Wert oder ein Flag auf den Stapel zu legen, da **ENDCASE** grundsätzlich den "Top of Stack" entfernte.
In der verbesserten Version bereinigt **ELSECASE** den Stapel. **ELSECASE** muß jedoch nicht aufgerufen werden; in diesem Fall kompiliert **ENDCASE** wie bisher ein DROP. Es ist jetzt möglich, zwischen den Worten **ELSECASE** und **ENDCASE** - wie auch zwischen **OF** und **ENDOF** - einen Wert auf den Stapel zu legen und diesen außerhalb der CASE-Kontrollstruktur zu verwenden.

Änderung:

Die Vorwärtsreferenzen werden nicht über den Stack aufgelöst, sondern über eine verkettete Liste.
Die Variable **caselist** enthält die Startadresse für noch nicht bekannte Sprungadressen. Die Schachtelungstiefe mehrerer CASE-Konstruktionen ist beliebig und wird durch initlist gelöst. **>marklist** füllt zur Kompilierzeit die Liste der Vorwärtsreferenzen und **>resolvelist** löst sie wieder auf.

Anwendungshinweis:

Wenn diese Definitionen außerhalb der Zusammenstellung des Arbeitssystems zugeladen werden, sollten nach dem Compilieren die Namen der mit ¦ als headerless markierten Worte mit **clear** entfernt werden.

Das Beispiel einer Tastatuabfrage auf CTRL-Tasten zeigt , wie dieses CASE-Konstrukt einzusetzen ist. Wichtig ist hierbei, daß das **OF** selbst die Gleichheit der beiden vorliegenden Werte prüft und in diesem Fall die Anweisungen zwischen **OF** und **ENDOF** ausführt.

```
: Control       bl word 1+ c@  $BF and  state @
                IF  [compile] Literal  THEN
; immediate

: Tastaturabfrage
  ." exit mit ctrl x" cr
  BEGIN key
        CASE control A OF ." action ^a " cr false ENDOF
             control B OF ." action ^b " cr false ENDOF
             control C OF ." action ^c " cr false ENDOF
             control D OF ." action ^d " cr false ENDOF
             control X OF ." exit "         true  ENDOF
        ELSECASE
                   ." befehl unbekannt " cr false
        ENDCASE
  UNTIL   ;
```

Mit dieser CASE-Anweisung läßt sich die Zuordnung der sechs Möglichkeiten zu den sechs Anweisungen ähnlich wie in PASCAL schreiben, lediglich Bereiche wie Ø..255 als Fall-Konstanten sind nicht erlaubt.

```
: Auswertung.5  ( Augenzahl -- )
      CASE
          1 OF  nehmen           ENDOF
          2 OF  links schieben   ENDOF
          3 OF  links schieben   ENDOF
          4 OF  rechts schieben  ENDOF
          5 OF  rechts schieben  ENDOF
          6 OF  trinken  ENDOF
      ELSECASE
          invers ." Betrug!" normal
      ENDCASE
;
```

Das vollständige Programm kann so geschrieben werden, wobei die typische Dreiteilung Eingabe-Verarbeitung-Ausgabe deutlich wird:

```
: craps   ( -- )

    cr Anfrage cr
       input#
       Auswertung
    cr Glückwunsch
;
```

Wil Baden hat in [1] ausgeführt, das eine CASE-Anweisung nur syntaktischer Zucker für ein Programm ist und letztendlich nichts weiter ist, als das Compilieren einer verschachtelten IF...THEN-Anweisung.
Eine solche Implemetierung für das volksFORTH83 wurde von Herrn K. Schleisiek-Kern geschrieben:

```
\ CASE   OF  ENDOF   ENDCASE     BREAK

 : CASE      ( n1 -- n1 n1 )  dup ;

 : OF        [compile] IF compile drop ;  immediate restrict

 : ENDOF     [compile] ELSE 4+ ;  immediate restrict

 : ENDCASE   compile drop
             BEGIN
               3 case?
             WHILE
               >resolve
             REPEAT ;  immediate restrict
```

Wil Badens Implementierung hält sich sehr eng an die logischen Grundlagen, wobei der Unterschied zum EAKER-CASE hauptsächlich darin besteht, daß hier jedes TRUE-Flag den Anweisungsteil zwischen OF und ENDOF ausführt; das OF nimmt keine Prüfung auf Gleichheit vor, sondern beliebige Ausdrücke können zu einem Flag führen, das dann von OF ausgewertet wird. So ist das Auswerten des Fall-Index variabler als beim EAKER-CASE:

```
: Auswertung.6  ( Augenzahl -- )
   dup
      1 6 between  not
            IF invers ." Betrug!" normal  drop exit  THEN
   CASE 1 =  OF nehmen          ENDOF
   CASE 6 =  OF trinken         ENDOF
   CASE 4 <  OF links schieben  ENDOF
   CASE 3 >  OF rechts schieben ENDOF
   ENDCASE  ;
```

Hier bei dieser Konstruktion steht die Plausibilitätsprüfung ganz vorn, um den ELSECASE-Fall durch ein EXIT aus dem Wort zu erreichen. Wird keines der Worte aus der Auswahl-Liste ausgeführt, läßt sich mit BREAK eine andere Lösung erreichen:

```
: BREAK     compile exit
          [compile] THEN ;  immediate restrict
```

Dadurch, daß BREAK ein EXIT aus dem Wort darstellt, wird ein (implizites) ELSECASE erreichen, indem man die Anweisungen der Auswahl-Liste mit OF und BREAK klammert und die Anweisungen für den ELSE-Fall nach ENDCASE aufführt:

```
: Auswertung.7  ( Augenzahl -- )

   CASE 1 =          OF  nehmen           BREAK
   CASE 2 = 3 =or    OF  links schieben   BREAK
   CASE 4 = 5 =or    OF  rechts schieben  BREAK
   CASE 6 =          OF  trinken          BREAK
   ENDCASE
   invers ." Betrug!" normal  ;
```

4.3.2.2 Positionelles CASE

Eine ganz anderen Lösungsansatz bietet ein positioneller CASE Konstrukt, bei dem die Fallunterscheidung durch den Fall-Index tabellarisch vorgenommen wird.

Bei den bisherigen Lösungen wurden immer eine Reihe von Vergleichen zwischen einem Fall-Index und einer Liste von Fall-Konstanten vorgenommen; nun wird der Fall-Index selbst benutzt, die gewünschte Prozedur auszuwählen. Die Verwendung des Fall-Index als Selektor bringt auch Vorteile in der Laufzeit, da die Vergleiche entfallen.

Wenn FORTH-Worte in Tabellen abgelegt werden sollen, stellt sich das Problem, daß ein FORTH-Wort bei seinem Aufruf normalerweise die eincompilierten Worte ausführt.
Bei einer Tabelle ist das nicht erwünscht; dort ist sinnvollerweise gefordert, daß die Startadresse der Tabelle übergeben wird, um der Fall-Index als Offset in diese Tabelle zu nutzen.

Dies läßt sich in volksFORTH entweder auf die traditionelle Weise mit] und [oder dem volksFORTH-spezifischen Create: lösen:

```
Create  Glas
      ]  nehmen  links schieben
         rechts schieben  trinken  [

Create: Glas
         nehmen
         links schieben
         rechts schieben
         trinken ;
```

Diese Tabelle Glas macht auch deutlich, welche Funktion das Dummy-Wort **schieben** außer einer besseren Lesbarkeit noch hat: Es löst die Schwierigkeit, daß 6 möglichen Wurfergebnissen nur 4 mögliche Aktionen gegenüberstehen.

Die Art und Weise des Zugriffs in **BEWEGEN** entspricht dem Zugriff auf eine Zahl in einem eindimensionalen Feld, einem Vektor:

```
: bewegen  ( addr n -- cfa )
    2* +  perform ;

: richtig  ( n  --  0<= n <= 3 )
    swap
    1 max  6 min
    3 case? IF  2  1- exit  THEN
    5 case? IF  4  1- exit  THEN
    1-  ;
```

Dieses Wort **RICHTIG** läßt zwar Werte kleiner als 1 und größer als 6 zu, justiert sie aber auf den Bereich zwischen 1 und 6 . Auch hier müßte eine Möglichkeit geschaffen werden, ein Wurfergebnis außerhalb der 6 Möglichkeiten als Betrugsversuch zurückzuweisen !
Die Verbindung von Tabelle und Zugriffsprozedur wird von dem Wort **:Does>** vorgenommen:

```
\ :Does>  für  Create <name> :Does> <action> ;   ks 25 aug 88

! : (does>   here >r [compile] Does> ;

  : :Does>    last @ 0= Abort" without reference"
     (does>  current @  context !   hide 0 ] ;
```

Dieses Wort **:DOES>** weist dem letzten über Create definierten Wort einen Laufzeit-Teil zu. Dieses Wort wurde von Herrn K. Schleisiek-Kern programmiert ; auch hier gilt der Hinweis, nach dem Compilieren das mit ! als headerless deklarierte Wort durch **clear** zu löschen.

```
Create: Auswertung.8
          nehmen
           links schieben
           rechts schieben
          trinken ;
       :Does>
          richtig bewegen ;
```

Ohne :DOES> sind die Tabelle und die Zugriffsprozeduren voneinander unabhängige Worte:

```
: CRAPS1
     cr  Anfrage cr
           input#
           Glas richtig bewegen
     cr  Glückwunsch  ;
```

Entschließt man sich dagegen, sowohl Tabelle als auch Zugriffsprozedur in einem Wort zu definieren, so ergibt sich das gewohnte Erscheinungsbild:

```
: CRAPS
     cr  Anfrage cr
           input#
           Auswertung
     cr  Glückwunsch  ;
```

Bei häufigerem Einsatz solcher Tabellen bietet sich der Einsatz von positional CASE defining words an. Auch hier wiederum zuerst die volks4TH-gemäße Lösung, danach die traditionelle Variante:

```
: Case:   ( -- )
   Create:  Does>   ( pfa -- )  swap 2* +  perform ;

\ alternative Definition für CASE:
: Case:
        :  Does>   ( pfa -- )  swap 2* +  perform ;
```

Eine sehr elegante Möglichkeit, die Fehlerbehandlung im Falle eines unglaubwürdigen Fall-Indexes zu handhaben, bietet das Wort Associative: .
Dieses Wort Associative: durchsucht eine Tabelle nach einer Übereinstimmung zwischen einem Zahlenwert auf dem Stack und den Zahlenwerten in der Tabelle und liefert den Index der gefundenen Zahl (match) zurück. Im Falle eines Mißerfolgs (mismatch) wird der größtmögliche Index +1 (out of range = maxIndex +1) übergeben:

```
: Associative:          ( n -- )
   Constant   Does>     ( n - index )
   dup @ -rot
   dup @ 0
   DO  2+  2dup  @ =
       IF  2drop drop  I  0 0  LEAVE  THEN
   LOOP 2drop ;


6 Associative:  Auswerten

                  1 ,
                2 , 3 ,
                4 , 5 ,
                  6 ,

          Case:  Handeln          \ besteht aus :

                  nehmen
```

```
            links  links
          rechts  rechts
              trinken
        schimpfen ;
```

Statt der Primitivabsicherung über **MIN** und **MAX** wird eine out of range Fehlerbehandlung namens **schimpfen** an der Tabellenposition maxIndex +1 durchgeführt.

4.3.2.3 Einsatzmöglichkeiten

Dieser letzte Teil der Ausführungen über die Möglichkeiten, eine CASE-Situation zu handhaben, greift Anregungen aus der Literatur [5],[6] auf.

Dazu werden zwei Worte definiert:

CLS löscht den gesamten Bildschirm und
CELLS macht die Berechnung des Tabellenzugriffs deutlicher:

```
: cls    full page ;
: cells  2* ;
```

Das Inhaltliche und die tabellarische Struktur bleiben unverändert, lediglich die Behandlung einer out of range Situation wird diesmal mit **min** und **max** und zweimaligem Eintragen der Fehler-Routine **schimpfen** verwirklicht.

```
Create: Handlung
          schimpfen   nehmen links links
                      rechts rechts trinken   schimpfen  ;

\ Die Ausführung einer Liste  nach Floegel 7/86
: auswählen   ( addr n -- *cfa ) 2 arguments
     swap  0 max                \ out of range MIN
           7 min                \ out of range MAX
     cells +  ;

: auswerten   ( n -- ) 1 arguments
     Handlung auswählen  perform ;

: .all  ( -- )
     8 0  DO  cr I dup . auswerten  2 spaces  LOOP ;
```

AUSWÄHLEN übergibt bei gegebenem Vektor und gegebenem Index einen Zeiger auf die code field address des entsprechenden Wortes. **AUSWERTEN** führt das so ausgewählte Wort aus und **.ALL** diente nur zur Kontrolle. Solch ein Wort, das angelegte Datenstrukturen auf dem Bildschirm darstellt, sollte in der Entwicklungsphasen eines Programmes immer dabei sein.

Eine weitere Möglichkeit, Werte in einen Vektor einzutragen, hat Herr Floegel in seinem Buch [4] dargestellt:

```
Create Tabelle  8 cells allot
       :Does> ( i -- addr ) swap cells + ;

 ' schimpfen  0 Tabelle !
 ' nehmen     1 Tabelle !
```

```
' links dup  2 Tabelle !
             3 Tabelle !
' rechts dup 4 Tabelle !
             5 Tabelle !
' trinken    6 Tabelle !
' schimpfen  7 Tabelle !

: auswerten  ( i -- )  0 max  7 min  Tabelle perform ;

: .action    ( i -- )
     Tabelle @  >name  bright .name normal ;
: .Tabelle  ( -- )  cr  8 0 DO  cr I .action  LOOP ;
```

Hier besteht mit .ACTION und .TABELLE die Möglichkeit, sich den Vektor darstellen zu lassen. In ähnlicher Weise werden auch im Kommandozeilen-Editor CED die neuen Aktionen in die Eingabe-Vektoren eingetragen.

Eine geringfügige Modifikation von [5] soll die Verknüpfung eines Vektors von Worten und einer Menü-Option zeigen:

```
Create function
        ] noop noop noop noop
          noop noop noop noop [
        :Does> ( i -- addr )
          swap  0 max 7 min   cells + ;
```

function ist ein execution vector , der mit NOOP vorbesetzt ist. Zur Laufzeit liefert er die Adresse des indizierten Elementes zurück.

```
: .action  ( i addr -- )
     @   >name  bright .name normal ;
```

.WORD gibt den Namen eines Wortes aus, dessen CFA in eine Adresse eingetragen wurde.

```
: option   ( i -- )
     r>
     dup 2+ >r                  \ i *w.addr
     @                          \ i w.addr
     stash swap  function !     \ i  w.addr i addr
     function .action ;         \ i addr
```

option holt die Adresse des auf **option** folgenden Wortes. Das Wort soll nicht ausgeführt werden, sondern das nachfolgende. Nur der Pointer auf das Wort soll ausgewertet werden. Nach dem übergebenen Index wird der Pointer in **function** eingetragen. Der Name des so eingetragenen Wortes wird angezeigt !

```
\ Menü        jrg 06feb89
: Menü
    0 option schimpfen
      1 option nehmen
      2 option links
      3 option links
      4 option rechts
      5 option rechts
      6 option trinken
    7 option schimpfen  ;
```

Wenn das Wort **MENÜ** aufgerufen wird, werden nicht nur die Optionen in die Tabelle eingetragen, sondern auch namentlich auf dem Bildschirm dargestellt. Diese Technik bietet sich für eine Menüzeile an fester Bildschirmposition an, ähnlich der Statuszeile des volksFORTH. Zum Ändern solcher Menüpunkte bieten sich die Funktionstasten an:

```
: fkey   ( -- )
   key &58 + abs  function perform ;
```

FKEY liefert beim Druck einer Funktionstaste einen Wert von -59 bis -68 zurück. Dieser wird für 1Ø Funktionstasten in den Bereich von -1 bis -1Ø skaliert und der Absolutwert gebildet.

[1] Wil Baden — Ultimate CASE-Statement (VD2/87 S.4Ø ff.)

[2] Dr. Charles Eaker — Just in CASE (FORTH DIM II/3)

[3] R. Zech — FORTH 83 (S.98ff/S.318f.)

[4] E. Floegel — FORTH Handbuch (S.1Ø9)

[5] W. Wejgaard — Menus in FORTH Elektroniker 9/88 (S.1Ø9 ff.)

4.4 Rekursion

Bevor die Technik der Rekursion für das volks4TH dargestellt wird, soll ein anderes Wort **.LASTNAME** zeigen, daß das Wort **LAST** mit dem in der Literatur oft anzutreffenden **LATEST** identisch ist:
Beide Worte liefern die name field address des zuletzt definierten Wortes im **CURRENT**-Vokabular. Das Wort **LAST'** dagegen liefert die cfa des zuletzt definierten Wortes.

```
: .lastname   last @  .name ;
```

Die Rekursion ist eine Technik, bei der ein Wort sich immer wieder selbst aufruft. Eines der bekannten Beispiele dafür ist die Berechnung der Fakultät einer positiven ganzen Zahl. Hierbei ergibt sich n! aus dem Produkt aller ihrer Vorgänger.

Im volks4TH ist der Selbstaufruf eines Wortes durch **RECURSIVE** gekennzeichnet, so daß sich ein Programm zur Fakultätsberechnung wie folgt präsentiert:

```
: fakultät  ( +n -- n! )
    recursive
    dup  0< IF  drop ." keine negativen Argumente! " exit
           THEN
    ?dup 0= IF  1  \ Spezialfall: 0
           ELSE  dup 1-  fakultät *
```

```
            THEN  ;

cr  4 fakultät .
cr  5 fakultät .
cr  6 fakultät .
```

Allerdings findet sich - vor allem in der figFORTH-Literatur - ein Wort **MYSELF**, das mit dem in FORTH83-Umgebungen anzutreffenden **RECURSE** identisch ist. Da auch diese Konstruktion, bei der **MYSELF/RECURSE** als Platzhalter für den Wortnamen dienen, gerne eingesetzt wird, werden die möglichen Definitionen und eine weitere Form von **FAKULTÄT** gezeigt:

```
: myself   last @  name> , ;  immediate

: myself   last' , ;  immediate

: recurse   [compile] myself ;  immediate
' myself  Alias  recurse   immediate

: fakultät  ( +n -- n! )
   dup 0< IF  ." keine negativen Argumente erlaubt! "
          ELSE  ?dup 0= IF  1
                        ELSE  dup 1-  myself *
                        THEN
          THEN  ;
```

Bei der Verwendung von **RECURSE** wird lediglich **MYSELF** dadurch ersetzt:

```
...
                ?dup 0= IF  1          \ Spezialfall: 0
                        ELSE  dup 1-  recurse *
                        THEN

...
```

5. Ein- / Ausgabe im volksFORTH

5.1 Ein- / Ausgabebefehle im volksFORTH

Alle Eingabe- und Ausgabeworte (**KEY EXPECT EMIT TYPE** etc.) sind im volks-FORTH vektorisiert, d.h. bei ihrem Aufruf wird die Codefeldadresse des zugehörigen Befehls aus einer Tabelle entnommen und ausgeführt. So ist im System eine Tabelle mit Namen **DISPLAY** enthalten, die für die Ausgabe auf dem Bildschirmterminal sorgt.

Dieses Verfahren der Vektorisierung bietet entscheidende Vorteile:

- Mit der Input-Vektorisierung kann man z.B. mit einem Schlag die Eingabe von der Tastatur auf ein Modem umschalten.
- Durch die Output-Vektorisierung können mit einer neuen Tabelle alle Ausgaben auf ein anderes Gerät (z.B. einenDrucker) geleitet werden, ohne die Ausgabebefehle selbst ändern zu müssen.
- Mit einem Wort (**DISPLAY, PRINT**) kann das gesamte Ausgabeverhalten geändert werden. Gibt man z.B. ein:

  ```
  print 1 list display
  ```

  wird Screen 1 auf einen Drucker ausgegeben und anschließend wieder auf den Bildschirm zurückgeschaltet. Man braucht also kein neues Wort, etwa **PRINTERLIST**, zu definieren.

Eine neue Tabelle wird mit dem Wort **OUTPUT:** erzeugt. Die Definition können Sie mit **view output:** nachsehen. **OUTPUT:** erwartet eine Liste von Ausgabeworten, die mit **;** abgeschlossen werden muß.

Beispiel:

```
Output: >PRINTER
   pemit pcr ptype pdel ppage pat pat? ;
```

Damit wird eine neue Tabelle mit dem Namen **>PRINTER** angelegt. Beim späteren Aufruf von **>PRINTER** wird die Adresse dieser Tabelle in die Uservariable **OUTPUT** geschrieben. Ab sofort führt **EMIT** ein **PEMIT** aus, **TYPE** ein **PTYPE** usw.

Die Reihenfolge der Worte nach OUTPUT:

```
userEMIT userCR userTYPE userDEL userPAGE userAT userAT?
```

muß unbedingt eingehalten werden.

Entsprechend wird die Input-Vektorisierung gehandhabt.

5.2 Ein- / Ausgaben über Terminal

Das volks4th verfügt über eine Reihe von Konstanten, die der besseren Lesbarkeit dienen:

c/row (-- Anzahl)
ist die Konstante, die die Anzahl der Zeichen pro Zeile (&80) angibt.

c/col (-- Anzahl)
ist die Konstante, die die Anzahl der Zeichen pro Spalte (&25) angibt.

c/dis (-- Anzahl.Cells)
ist die Konstante, die die Größe des Speichers für einen Ausgabeschirm angibt.

c/l (-- +n) "characters-per-line"
+n ist die Anzahl der Zeichen pro Screenzeile. Aus historischen Gründen ist dieser Wert &64 bzw. $40 .

l/s (-- +n) "lines-per-screen"
+n ist die Anzahl der Zeilen pro Screen.

bl (-- n) "b-l"
n ist der ASCII-Wert für ein Leerzeichen.

#esc (-- n) "number-escape"
n ist der ASCII-Wert für Escape.

#cr (-- n) "number-c-r"
n ist der Wert, den man durch KEY erhält, wenn die Return-Taste <cr> gedrückt wird.

#lf (-- n) "number-linefeed"
n ist der ASCII-Wert für Linefeed.

#bel (-- n)
n ist der ASCII-Wert für BELL.

#bs (-- n) "number-b-s"
n ist der Wert, den man durch KEY erhält, wenn die Backspace-Taste gedrückt wird.

standardi/o (--) "standard-i-o"
stellt sicher, daß die beim letzten save bestimmten Ein- und Ausgabegeräte wieder eingestellt sind.

inputkol und **outputkol**
sind beides definierende Wörter, die eine festgelegte Anzahl von Zeigern auf Prozeduren erwarten.

area (-- addr)
ist eine Uservariable und zeigt auf den Zustandsvektor des aktiven Windows.

areakol
ist ein definierendes Wort, das zur Erzeugung eines Windows benutzt wird.

terminal
ist ein Window, das sich von Zeile Ø - 23 erstreckt für das Terminal Output Window.

window (topline bottomline --)
setzt die oberste und unterste Zeile des aktuellen Windows neu.

full (--)
setzt das aktuelle Window über den ganzen Bildschirm von der obersten bis zur untersten Zeile. Somit entspricht

```
: cls   full page ;
```

dem Befehl, der den gesamten Bildschirm löscht.

curat? (-- row col)
liefert die aktuelle Cursorposition im aktiven Fenster.

cur! (--)
setzt den Cursor in das Window, auf das AREA zeigt.

setpage (n --)
setzt aktive Bildschirmseite.

video@ (-- seg)
liefert das Segment zurück, in dem der Speicher der Videokarte liegt.

savevideo (-- seg ¦ ff)
rettet den Speicherbereich der Videoausgabe; wird bei dem Wort **MSDOS** eingesetzt.

restorevideo (seg --)
restauriert die Bildschirmausgabe auf dem angegebenen Segment; wird bei dem Wort **MSDOS** eingesetzt.

catt (-- addr)
ist eine Variable, die das Attribut zur Zeichendarstellung enthält, z.B. das Zeichenattribut **INVERS** .

list (u --) 83
zeigt den Inhalt des Screens u auf dem aktuellen Ausgabegerät an. **SCR** wird auf u gesetzt.
Siehe **BLOCK** .

(page (--) "(page"
löscht den Bildschirm und positioniert den Cursor in die linke obere Ecke.
Vergleiche **PAGE** .

page (--)
bewirkt, daß der Cursor des Ausgabegerätes auf eine leere neue Seite bewegt wird. Eines der OUTPUT-Worte.

(del (--) "(del"
löscht ein Zeichen links vom Cursor.
Vergleiche **DEL** .

del (--)
löscht das zuletzt ausgegebene Zeichen. Bei Druckern ist die korrekte Funktion nicht immer garantiert. Eines der OUTPUT-Worte

(cr (--) "(c-r"
setzt den Cursor in die erste Spalte der nächsten Zeile. Ein **PAUSE** wird ausgeführt.

cr (--) 83 "c-r"
bewirkt, daß die Schreibstelle des Ausgabegerätes an den Anfang der nächsten Zeile verlegt wird. Eines der OUTPUT-Worte.

?cr (--) "question-c-r"
prüft, ob in der aktuellen Zeile mehr als C/L Zeichen ausgegeben wurden und führt in diesem Fall CR aus.

(at (row col --) "(at"
setzt die aktuelle Cursorposition. (AT positioniert den Cursor in der Zeile row und der Spalte col . Ein Fehler liegt vor, wenn row > $18 (&24) oder col > $5Ø (&8Ø) ist. Die fehlerhafte Ausgabe wird nicht unterdrückt!
Vergleiche AT .

(at? (-- row col) "(at-question"
liefert die aktuelle Cursorposition. row ist die aktuelle Zeilennummer des Cursors, col die aktuelle Spaltennummer.
Vergleiche AT? .

at (row col --)
positioniert die Schreibstelle des Ausgabegerätes in die Zeile row und die Spalte col. AT ist eines der über OUTPUT vektorisierten Worte.
Siehe AT? .

.i.at? (-- row col) "at-question"
ermittelt die aktuelle Position der Schreibstelle des Ausgabegerätes und legt Zeilen- und Spaltennummer auf den Stack. Eines der OUTPUT-Worte.

col (-- #col) "col"
#col ist die Spalte, in der sich die Schreibstelle des Ausgabegerätes gerade befindet.
Vergleiche ROW und AT? .

row (-- #row) "row"
#row ist die Zeile, in der sich die Schreibstelle des Ausgabegerätes befindet.die Benutzung beschreiben-
Vergleiche COL und AT? .

curoff (--)
schaltet den Cursor aus.

curon (--)
schaltet den Cursor ein.

cursshape (topline bottomline --)
bestimmt das Aussehen des Cursors.

5.3 Drucker-Ausgaben

printer (--)
ist das Vokabular mit den Worten zur Druckersteuerung.

print (--)
schaltet die Ausgabe auf den Drucker um

+print (--)
schaltet den Drucker zusätzlich zu.

lst! (8b --)
gibt ein Byte zum Drucker aus.

5.4 Ein- / Ausgabe von Zahlen

Die Eingabe von Zahlen erfolgt im interpretativen Modus über die Tastatur, wobei grundlegend Eingabeworte mit **number** **number?** und den verwandten Worten definiert werden.
Bei der Ausgabe von Zahlen ist wieder die fehlende Typisierung von FORTH zu beachten - für ein bestimmtes Datenformat (integer, unsigned, double) ist jeweils der geeignete Operator auszuwählen.

. (n --)
gibt den obersten Stack-Werte als Zahl (integer) aus.
Soll die Ausgabe des nachfolgenden Leerzeichens unterdrückt werden, so sind die Befehle für rechtsbündige Zahlenausgabe .r u.r d.r mit einer Feldlänge von Ø zu benutzen.

u. (u --)
gibt den obersten Stack-Wert als vorzeichenlose 16Bit-Zahl (unsigned) aus.

d. (d --)
gibt die obersten beiden Stack-Werte als 32Bit-Zahl (double) aus.

.r (n Feldlänge --)
druckt eine 16Bit-Zahl in einem Feld mit angegebener Länge rechtsbündig aus.

u.r (u Feldlänge --)
gibt den obersten Stack-Wert als vorzeichenlose 16Bit-Zahl in einem Feld rechtsbündig aus.

d.r (d Feldlänge --)
gibt eine 32Bit-Zahl in einem Feld rechtsbündig aus.

5.5 Ein- / Ausgabe über einen Port

pc@ (port.addr -- 8b)
holt ein Byte von port.addr aus einem Peripheriebaustein des 8Ø86-Systems auf den Stack.

pc! (8b port.addr --)
speichert ein Byte in einen Peripheriebaustein des 8Ø86-Systems bei port.addr.

5.6 Eingabe von Zeichen

In FORTH wird man immer einen Speicherbereich benennen, in dem Zeichen und Zeichenketten verarbeitet werden. Hierfür verwendet man meistens einen kleinen, 8Ø Zeichen langen Speicherbereich namens PAD . Dieser Notizblock - so die deutsche Übersetzung von pad - belegt keinen festen Speicherbereich und steht sowohl dem FORTH-System als auch dem zur Verfügung.

Dann möchte ich Ihnen mit dem Texteingabe-Puffer tib einen weiteren wichtigen Speicherbereich vorstellen,der den vernünftigen Umgang mit den angeschlossenen Geräten sicherstellt. Weil die Texteingabe über die Tastatur relativ langsam vor sichgeht, werden die Zeichen hier erst in einem freien Speicherbereich, dem Pufferspeicher tib , gesammelt und dann abgearbeitet.

tib (-- addr) 83
liefert die Adresse des Text-Eingabe-Puffers. Hier wird die Eingabe-Befehlszeile des aktuellen Eingabegerätes (meist KEYBOARD) gespeichert.
Siehe >TIB .

#tib (-- addr) 83 "number-t-i-b"
addr ist die Adresse einer Variablen, die die Länge des aktuellen Textes (die Anzahl der Zeichen) im Text-Eingabe-Puffer enthält.
Vergleiche TIB .

>tib (-- addr) "to-tib"

addr ist die Adresse eines Zeigers auf den Text-Eingabe-Puffer.
Siehe **TIB** .

>in (-- addr)

ist eine Variable, die den Offset auf das gegenwärtige Zeichen im Quelltext enthält.
>in indiziert relativ zum Beginn des Blockes, der durch die Variable **blk** gekennzeichnet wird. Ist blk = Ø , so wird als Quelle der **tib** angenommen.
Vergleiche **word** .

pad (-- addr)

liefert die Adresse des temporären Speichers **PAD** . Die Adresse von **PAD** ändert sich, wenn der Dictionarypointer **DP** verändert wird, z.B. durch **ALLOT** oder , (Komma).

input (-- addr) U

addr ist die Adresse einer Uservariablen, die einen Zeiger auf ein Feld von vier Kompilationsadressen enthält, die für ein Eingabegerät die Funktionen **KEY KEY? DECODE** und **EXPECT** realisieren.
Vergleiche die Beschreibung der INPUT- und OUTPUT-Struktur.

keyboard (--)

ein mit **INPUT:** definiertes Wort, das die Tastatur als Eingabegerät setzt. Die Worte **KEY KEY? DECODE** und **EXPECT** beziehen sich auf die Tastatur.
Siehe **(KEY (KEY? (DECODE** und **(EXPECT** .

empty-keys (--)

löscht den Tastaturpuffer über den Int.21h, Fct.ØC, AL=Ø.

(key? (-- flag) "(key-question"

Das Flag ist **TRUE** , wenn eine Taste gedrückt wurde, sonst **FALSE** .
Vergleiche **KEY?** .

key? (-- flag) "key-question"

Das Flag ist **TRUE** , falls ein Zeichen zur Eingabe bereitsteht, sonst ist flag **FALSE** . Eines der INPUT-Worte.

(key (-- 16b) "(key"

wartet auf einen Tastendruck. Während der Wartezeit wird **PAUSE** ausgeführt.
Zeichen des erweiterten ASCII-Zeichensatzes werden in den unteren 8 Bit von 16b übergeben. Funktionstasten liefern negative Werte.Steuerzeichen werden nicht ausgewertet, sondern unverändert abgeliefert.
Vergleiche **KEY** .

key (-- 16b) 83

empfängt ein Zeichen vom Eingabegerät. Es wird kein Echo ausgesandt. **KEY** wartet, bis tatsächlich ein Zeichen empfangen wurde.
Die niederwertigen 8 Bit enthalten den ASCII-Code des zuletzt empfangenen Zeichens. Alle gültigen ASCII-Codes können empfangen werden.
Die oberen 8 Bit enthalten systemspezifische Informationen, den Tastatur-Scancode.
Die Funktionstasten werden als negative Zahlenwerte zurückgegeben, so daß beispielsweise auf einer PC/XT-Tastatur folgende Tasten entsprechende 16Bit-Werte liefern :

Tasten F1 - F10:	-59 - -68
Cursorblock:	-71 (home) - -81 (pgdn)
Zahlen 0 - 9 :	48 (0) - 57 (9)

KEY ist eines der INPUT-Worte.

(decode (addr pos0 key -- addr pos1) "(decode"

wertet **key** aus. **key** wird in der Speicherstelle addr+pos1 abgelegt und als Echo auf dem Bildschirm ausgegeben. Die Variable **SPAN** und pos werden inkrementiert. Backspace löscht das Zeichen links vom Cursor und dekrementiert pos1 und **SPAN** .
Vergleiche **INPUT:** und **(expect** .

(expect (addr len --) "(expect"

erwartet len Zeichen vom Eingabegerät, die ab addr im Speicher abgelegt werden. Ein Echo der Zeichen wird ausgegeben. Return beendet die Eingabe vorzeitig. Ein abschließendes Leerzeichen wird statt des <CR> ausgegeben. Die Länge der Zeichenkette wird in der Variablen **SPAN** übergeben.
Vergleiche **EXPECT** .

expect (Zieladresse maxAnzahl --) 83

empfängt maxAnzahl Zeichen und speichert sie ab der Zieladresse im Speicher, ohne ein count byte einzubauen. Der count wird statt dessen in der Variablen **span** abgelegt. Ist maxAnzahl = Ø , so werden keine Zeichen übertragen.
Die Übertragung wird beendet, wenn ein #CR erkannt oder maxAnzahl Zeichen übertragen wurden. Das #CR wird nicht mit abgespeichert.
Alle Zeichen werden als Echo, statt des #CR wird ein Leerzeichen ausgegeben.
Dies ist das **expect** aus der Literatur, mit dem man an beliebigen Stellen im Speicher Eingabepuffer jeder Länge anlegen kann. Eines der INPUT-Worte.
Vergleiche **SPAN** .

span (-- addr) 83

Der Inhalt der Variablen **SPAN** gibt an, wieviele Zeichen vom letzten **EXPECT** übertragen wurden; in ihr ist die Anzahl der tatsächlich eingegebenen Zeichen des zuletzt ausgeführten **expect** abgelegt.
span wird ausgelesen, um zu sehen, ob die erwartete Zeichenzahl empfangen oder die Eingabe vorher mit <CR> abgebrochen wurde. Bei **span** ist zu beachten, daß es direkt nach einem **expect** benutzt werden muß, weil auch einige FORTH-Systemwörter **span** benutzen.

>expect (Zieladresse maxAnzahl.Zeichen --)

legt ebenfalls einen zu erwartenden String an einer Zieladresse ab, allerdings wird jetzt das count-byte eingebaut.

nullstring? (addr -- addr ff ¦ addr tf)

prüft, ob der counted String an der gegebenen Adresse die Länge NULL hat. Wenn dies der Fall ist, wird **true** hinterlegt, ansonsten bleibt addr erhalten und **false** wird übergeben.

stop? (-- flag) "stop-question"

Ein komfortables Wort, das es dem Benutzer gestattet, einen Programmablauf anzuhalten oder zu beenden.
Steht vom Eingabegerät ein Zeichen zur Verfügung, so wird es eingelesen. Ist es #ESC oder CTRL-C , so ist Flag **TRUE** , sonst wird auf das nächste Zeichen gewartet. Ist dieses jetzt #ESC oder CTRL-C , so wird **STOP?** mit **TRUE** verlassen, sonst mit **FALSE** .
Steht kein Zeichen zur Verfügung, so ist Flag **FALSE** .

source (-- addr len)

übergibt Adresse und Länge des Quelltextes; wenn die Blocknummer = Ø ist, entsprechen diese Angaben denen des Text-Eingabe-Puffers TIB ; das ist der Grund, warum Block Ø niemals geladen werden kann. Ist die Blocknummer nicht NULL, beziehen sich die Angaben auf den Massenspeicher.

word (delim -- addr)

durchsucht den Quelltext (siehe SOURCE) nach einem frei wählbaren Begrenzer und legt den gefundenen String in der counted Form im Speicher an der Adresse here ab.
Führende Zeichen vom Typ delim werden ignoriert. Deshalb erwartet word immer einen Delimiter und übernimmt nicht automatisch den üblichen Begrenzer, das Leerzeichen. Daher typisch:

```
... bl word ...
```

parse (delim -- addr len)

erwartet ebenfalls ein Zeichen als Begrenzer, liefert die nächste delim begrenzte Zeichenkette des Quelltextes aber schon in der üblichen Form addr count passend für type.
Die Länge ist Null, falls der Quelltext erschöpft oder das erste Zeichen der delimiter ist. **parse** verändert die Variable >in .
Typisches Auftreten:

```
: .(   ASCII ) parse  type ;
```

name (-- string)

holt den nächsten durch Leerzeichen abgeschlossenen String aus dem Quelltext, wandelt ihn in Großbuchstaben um und hinterläßt die Adresse, ab der der String im Speicher steht.
Siehe **word** .

find (string -- string ff | cfa tf)

erwartet die Adresse eines counted Strings. Dieser String wird in der aktuellen Suchreihenfolge gesucht und das Resultat dieser Suche übergeben.
Wird das Wort gefunden, so liegt die cfa und ein Pseudoflag tf bereit:
- Ist das Wort immediate, so ist das flag positiv, sonst negativ.
- Ist das Wort restrict, so hat das flag den Betrag 2, sonst Betrag 1.

Typisch:

```
... (name) find ...
... " <String>" find ...
... name  find ...
```

execute (addr --)

Das durch addr gekennzeichnete Wort wird aufgerufen bzw. ausgeführt. Dabei kann es sich um ein beliebiges Wort handeln. Eine Fehlerbedingung existiert, falls addr nicht cfa eines Wortes ist.
Typisch: ' <name> execute

perform (addr --)

ist ein execute für einen Vektor und entspricht @ execute . Typisch beim Einsatz einer Prozedurvariable:

```
Variable <vector>
' noop <vector> !
(vector) perform
```

Hier wird zuerst eine Variable angelegt und anschließend dieser Variablen mit Hilfe des ' ("Tick") die CFA eines Wortes zugewiesen. Dieses Wort kann nun über die Adresse der Prozedurvariablen und PERFORM ausgeführt werden. Allgemein wird diese Variable mit NOOP abgesichert.

query (--) 83

ist die allgemeine Texteingabe-Prozedur für den Texteingabe-Puffer. query stoppt die Programmausführung und liest eine Textzeile von max. 80 Zeichen mit Hilfe von expect ein.
Die Zeichen werden von einem Eingabegerät geholt und in den Text-Eingabe-Puffer übertragen, der bei TIB beginnt. Die Übertragung endet beim Empfang von #CR oder wenn die Länge des Text-Eingabe-Puffers erreicht wird. Der Inhalt von >IN und BLK wird zu 0 gesetzt und #TIB enthält die Zahl der empfangenen Zeichen.
Zu beachten ist, daß der Inhalt von TIB überschrieben wird. Damit werden auch alle Worte im TIB , die noch nicht ausgeführt wurden, ebenfalls überschrieben. Um Text aus dem Puffer zu lesen, kann WORD benutzt werden.
Siehe EXPECT .

interpret (--)

ist der allgemeine Text-Interpreter des FORTH-Systems und beginnt die Interpretation des Quelltextes bei dem Zeichen, das durch den Inhalt der Variablen >in bezeichnet wird. >in indiziert relativ zum Beginn des Blockes, der durch die Variable blk gekennzeichnet wird. Ist blk = 0 , so interpretiert interpret die max. 80 Zeichen im tib .

5.7 Ausgabe von Zeichen

output (-- addr) U

addr ist die Adresse einer Uservariablen, die einen Zeiger auf sieben Kompilationsadressen enthält, die für ein Ausgabegerät die Funktionen **EMIT CR TYPE DEL PAGE AT** und **AT?** realisieren.
Vergleiche die gesonderte Beschreibung der Output-Struktur.

display (--)

ist ein mit **OUTPUT:** definiertes Wort, das den Bildschirm als Ausgabegerät setzt, wenn es ausgeführt wird. Die Worte **EMIT CR TYPE DEL PAGE AT** und **AT?** beziehen sich dann auf den Bildschirm.

(emit (8b --) "(emit"

gibt 8b auf dem Bildschirm aus. Ein **PAUSE** wird ausgeführt. Alle Werte werden als Zeichen ausgegeben, Steuercodes sind nicht möglich, d.h. alle Werte < $2Ø werden als PC-spezifische Zeichen ausgegeben.
Vergleiche **EMIT** .

emit (16b --) 83

gibt die unteren 8 Bit an das Ausgabegerät. Eines der OUTPUT-Worte.

charout (8b --)

gibt 8b auf Standard-I/O-Gerät aus. Dazu wird die Fct.Ø6 des Int21h benutzt. ASCII-Werte < $2Ø werden als Steuercodes interpretiert. **CHAROUT** ist ein Primitiv für die Ausgabe-routinen und sollte vermieden werden, da die Ausgabe dann nicht über die Videotreiber läuft.

tipp (addr count --)

ist die Primitiv-Implementierung für **TYPE** über den MSDOS Character-Output.
Siehe auch **DISPLAY** in KERNEL.SCR.

(type (addr len --)

gibt den String, der im Speicher bei addr beginnt und die Länge len hat, auf dem Bildschirm aus. Ein **PAUSE** wird ausgeführt.
Vergleiche **TYPE OUTPUT:** und **(EMIT** .

type (addr +n --) 83

sendet +n Zeichen, die ab addr im Speicher abgelegt sind, an ein Ausgabegerät. Ist +n = Ø , so wird nichts ausgegeben.

ltype (seg:addr Länge --)
ist ein segmentiertes TYPE .

space (--)
gibt ein Leerzeichen aus.

spaces (Anzahl --)
gibt eine bestimmte Anzahl von Leerzeichen aus.

6. Strings

Hier befinden sich grundlegende Routinen zur Stringverarbeitung.
Vor allem wurden auch Worte aufgenommen, die den Umgang mit den vom Betriebssystem geforderten Ø-terminated Strings ermöglichen. FORTH hat hier gegenüber C den Nachteil, daß FORTH-Strings standardmäßig mit einem Count-Byte beginnen, das die Länge des Strings enthält. Ein abschließendes Zeichen (z.B. ein Null-Byte) ist daher unnötig. Da das Betriebssystem aber in C geschrieben wurde, müssen Strings entsprechend umgewandelt werden.
Standardmäßig arbeitet FORTH mit counted Strings, die lediglich durch eine Adresse gekennzeichnet werden. Das Byte an dieser Adresse enthält die Angabe, wie lang die Zeichenkette ist. Auf dieses count byte folgt dann die Zeichenkette selbst. Dadurch ist die Länge eines Standard-Strings in FORTH auf 255 Zeichen begrenzt.
Die kürzeste Zeichenkette ist ein String der Länge NULL, für dessen Überprüfung der Befehl **NULLSTRING?** zur Verfügung steht.

So sieht der String FORTH an der Adresse addr im Speicher unter FORTH aus.

." (--)

ist nur während des Compilierens in Wörtern zu verwenden. Die gewünschte Zeichenkette wird von ." und " eingeschlossen, wobei das Leerzeichen nach ." nicht mitzählt. Wenn das Wort abgearbeitet wird, so wird der String zwischen ." und " ausgegeben:

```
: Hallo ." Ich mache FORTH" ;
```

" (-- addr) C,I "string"
(--) compiling

liest den Text bis zum nächsten " und legt ihn als counted string im Dictionary ab. Kann nur während der Kompilation ver-wendet werden. Zur Laufzeit wird die Startadresse des counted string auf den Stack gelegt. Das Leerzeichen, das auf das erste Anführungszeichen folgt, ist nicht Bestandteil des Strings. Wird in der folgenden Form benutzt:

```
: Hallo " Ich mache FORTH" count type ;
```

," (--) "string literal"
speichert einen counted String ab **HERE** . Wird in der folgenden Form benutzt :
```
Create Hallo ," Ich mache FORTH"
Hallo count type
```

nullstring? (addr -- addr false ¦ true)
prüft, ob der counted String an der Adresse addr ein String der Länge NULL ist. Wenn dies zutrifft, wird **TRUE** übergeben. Anderenfalls bleibt addr erhalten und **FALSE** wird übergeben.

"lit (-- addr)
wird in compilierenden Worten benutzt, die auf INLINE-Strings zugreifen. Siehe auch **?"** und **"** .

.((--)
druckt den nachfolgenden String immer sofort aus. Der String wird von) beendet.
Anwendung: `.( Ich mache FORTH )`

((--)
leitet einen Kommentar ein, d.h. Text, der vom Compiler ignoriert wird.

) beendet einen Kommentar. Dieses Wort ist aber selbst kein FORTH-Befehl, muß deshalb auch nicht mit Leerzeichen abgetrennt sein.

6.1 String-Manipulationen

Hier im Glossar bezeichnet der Stackkommentar (string --) die Adresse eines counted Strings, dagegen (addr len --) die Charakterisierung durch die Anfangsadresse der Zeichenkette und ihre Länge.

Keine Stringvariable - Benutze:
```
: String: Create dup , 0 c, allot Does> 1+ count ;
```

caps (-- addr)
liefert die Adresse einer Variablen, die angibt, ob beim Stringvergleich auf Groß- und Kleinschreibung geachtet werden soll.

capital (char -- char')
Die Zeichen im Bereich a bis z werden in die Großbuchstaben A bis Z umgewandelt, ebenso die Umlaute äöü . Andere Zeichen werden nicht verändert.

upper (addr len --)

wandelt einen String in Großbuchstaben um. Es werden keine Parameter zurückgegeben.

capitalize (string -- string)

ist durch **UPPER** ersetzt worden. Bei Bedarf kann es folgendermaßen definiert werden:

```
: capitalize  ( string -- string )  dup count upper ;
```

/string (addrØ len index -- addr1 restlen) "cut-string"

verkürzt den durch addrØ und len gekennzeichneten String an der mit index angegebenen Stelle und hinterläßt die Parameter des rechten Teilstrings.

-trailing (addrØ lenØ -- addr1 len1) "minus-trailing"

kürzt einen String um die Leerzeichen, die sich am Ende des Strings befinden. Anschließend liegen die neueen Parameter auf dem Stack, passend z.B. für **TYPE** .

scan (addrØ len char -- addr1 restlen)

durchsucht einen mit addr und len angegebenen String nach einem Zeichen char.
Wird das Zeichen gefunden, so wird die Adresse der Fundstelle und die Restlänge einschließlich des gefundenen Zeichens übergeben.
Wird char nicht gefunden, so ist addr1 die Adresse des ersten Bytes hinter dem String und die Restlänge ist NULL.

skip (addrØ len char -- addr1 restlen)

durchsucht einen mit addrØ und len angegebenen String nach einer Abweichung von dem gegebenen Zeichen char.
Beim ersten abweichenden Buchstaben wird die Adresse der Fundstelle und die Restlänge ohne das abweichende Zeichen übergeben.
Besteht der gesamte String aus dem Zeichen char, so ist addr1 die Adresse des Bytes hinter dem String und die Restlänge ist NULL.

?" (--)

gibt die Position eines Zeichen in einem String an. Dieses ?" wird in Definitionen benutzt, um z.B. bei

```
: vocal?   ( char -- index )   capital ?" aeiou"  ;
Ascii ( char)  vocal?
```

festzustellen, ob es sich bei dem gegebenen Zeichen um einen Vokal handelt (index ungleich NULL) oder um welchen Vokal es sich handelt (a=1, e=2, i=3 etc.) .

count (string -- addr+1 len)

wandelt eine String-Addresse in die Form addr len um, die für z.B. TYPE notwendig ist.

bounds (addr len -- limit start)

ist definiert als

```
: bounds   over + swap ;
```

und kann z.B. dazu benutzt werden, die Parameter einer Zeichenkette in DO..LOOP-gemäße Argumente umzurechnen:

```
: upper ( addr len -- )
    bounds ?DO
             I c@ capital I c!
           LOOP ;
```

type (addr len --)

gibt eine Zeichenkette aus, die an addr beginnt und die angebene Länge hat.

>type (addr len --)

ist ein TYPE , das zuerst den String nach PAD kopiert. Dies ist in Multitasking-Systemen wichtig und wird folgendermaßen definiert:

```
: >type   ( addr count -- )
     pad place
     pad count type ;
```

place (addrØ len dest.addr --)

legt eine Zeichenkette, die durch ihre Anfangsadresse und ihre Länge gekennzeichnet ist, an der Zieladresse als counted String ab.
PLACE wird in der Regel benutzt, um Text einer bestimmten Länge als counted String abzuspeichern; dabei darf dest.addr gleich, größer oder kleiner als addrØ sein.
Die Wirkungsweise von PLACE wird in der Definition von >EXPECT deutlich, wo die Anzahl der von EXPECT eingelesenen Zeichen mit SPAN @ ausgelesen und von PLACE an der Zieladresse eingetragen wird:

```
: >expect ( addr len --)
     stash expect  span @  over place ;
```

attach (addr len string --)

fügt einen String an den durch addr und len gekennzeichneten String an, wobei der Count addiert wird.

append (char string --)

fügt ein Zeichen an den angegebenen String an und in-crementiert den Count.

detract (string -- char)
zieht das erste Zeichen aus einer Zeichenkette und de-crementiert den Count.

6.2 Suche nach Strings

6.2.1 In normalem Fließtext:

match (buf.addr buf.len string -- match.addr buf.rest)
Der Pufferbereich an buf.addr mit der Länge buf.len wird nach einer vollständigen Übereinstimmung mit dem counted String überprüft. Die Adresse der Übereinstimmung und die Restlänge des Puffers wird übergeben. **MATCH** befindet sich im Vokabular **EDITOR** .

search (buf len string -- offset flag)
durchsucht einen Speicherbereich mit angegebener Länge nach einem counted string.
Der Abstand vom Anfang des Speicherbereiches und ein Flag werden übergeben . Ist das Flag wahr, wurde der String gefunden, sonst nicht. Der Quelltext von **SEARCH** befindet sich im Editor und ist mit dessen Suchfunktion leicht aufzufinden.

6.2.2 Im Dictionary:

(find (string thread -- string ff ¦ nfa tf)
erwartet die Adresse eines counted string und eine Suchreihe. Wird das Wort in dieser Reihe gefunden, so wird ein true flag und die nfa übergeben; wird es nicht gefunden, wird ein faise flag übergeben und die Stringadresse bleibt erhalten.
Typische Anwendung:

```
... last @ current @ (find ...
```

find (string -- string.addr ff | cfa tf)

erwartet die Adresse eines counted Strings. Dieser String wird in der aktuellen Suchreihenfolge gesucht und das Resultat dieser Suche übergeben.
Wird das Wort gefunden, so liegt die cfa und ein Pseudoflag tf bereit:
- Ist das Wort immediate, so ist das Flag positiv, sonst negativ.
- Ist das Wort restrict, so hat das Flag den Betrag 2, sonst Betrag 1.

Typische Anwendung:

```
... ( name )  find ...
... " <string>" find ...
... name find ...
```

6.3 Ø-terminated Strings

Es gibt noch eine andere Darstellungsform für Strings, die beispielsweise für MS-DOS geeigneter ist. Diese Strings werden zwar ebenfalls durch eine Adresse gekennzeichnet; diese Adresse enthält aber kein count byte . Statt dessen werden diese Zeichenketten mit einem Nullbyte abgeschlossen.

asciz (-- asciz)
(--) compiling

holt das nächste Wort des Quelltextes (TIB oder BLOCK) in den Speicher und legt es als nullterminierten String (mit abschließendem Null-Byte) im Dictionary ab. Diese Adresse asciz im Dictionary wird zurückgeliefert.
Wird in der folgenden Form benutzt :

```
asciz <string>
```

Vergleiche ," .

>asciz (string addr.dest -- asciz)

wandelt den counted String an der Adresse STRING um in einen nullterminierten String, der an der gegebenen Ziel-Adresse ADDR abgelegt wird. Die Länge des Strings im Speicher bleibt gleich.
An der zurückgelieferten Adresse asciz liegt der neue String, wobei die übergebene Zieladresse und die zurückgelieferte ASCIZ-Adresse nicht gleich sein müssen.
>ASCIZ ist das grundlegende Wort, um FORTH-Strings in MS-DOS-Strings umzuwandeln.

counted (asciz -- addr len)

wird benutzt, um die Länge eines mit einem Nullbyte terminierten Strings zu bestimmen. asciz ist die Anfangsadresse dieses Strings, addr und len sind die Parameter, die z.B. von TYPE erwartet werden.

6.4 Konvertierungen: Strings -- Zahlen

6.4.1 String in Zahlen wandeln

digit? (char -- digit true)
(char -- false)

prüft, ob das Zeichen char eine gültige Ziffer entsprechend der aktuellen Zahlenbasis in **BASE** ist. Ist das der Fall, so wird der Zahlenwert der Ziffer und **TRUE** auf den Stack gelegt. Ist char keine gültige Ziffer, wird **FALSE** übergeben.

accumulate (+dØ addr char -- +d1 addr)

dient der Umwandlung von Ziffern in Zahlen.
Multipliziert die Zahl +dØ mit **BASE** , um sie eine Stelle in der aktuellen Zahlenbasis nach links zu rücken, und addiert den Zahlenwert von char. char muß eine in der Zahlenbasis gültige Ziffer darstellen, addr wird nicht verändert.
Wird z.B. in **CONVERT** benutzt. .

convert (+dØ addrØ -- +d1 addr1)

wandelt den Ascii-Text ab addrØ +1 in eine Zahl entsprechend der Zahlenbasis **BASE** um. Der enstehende Zahlenwert und die Adresse des ersten nicht wandelbaren Zeichens im Text werden hinterlassen.

number? (addr -- d Ø> ¦ n Ø< ¦ addr false)

wandelt den counted String bei der Adresse addr in eine Zahl n um. Die Umwandlung erfolgt entsprechend der Zahlenbasis in **BASE** oder wird vom ersten Zeichen im String bestimmt.
Enthält der String zwischen den Ziffer auch die Asciizeichen für Punkt oder Komma, so wird er als doppelt genaue Zahl interpretiert und Ø> gibt die Zahl der Ziffern hinter dem Punkt einschließlich an.
Sonst wird der String in eine einfach genaue Zahl n umgewandelt und eine Zahl kleiner als Null hinterlassen.
Wenn die Ziffern des Strings nicht in eine Zahl umgewandelt werden können, bleibt die Adresse des Strings erhalten und **FALSE** wird auf den Stack gelegt.

Die Zeichen, die zur Bestimmung der Zahlenbasis dem Ziffern-string vorangestellt werden können, sind:

```
%   ( Basis 2    "binär")
&   ( Basis 1Ø   "dezimal")
$   ( Basis 16   "hexadezimal")
h   ( Basis 16   "hexadezimal")
```

Der Wert in **BASE** wird dadurch nicht verändert.

number (addr -- d)

wandelt den counted String bei der Adresse addr in eine Zahl d um. Die Umwandlung erfolgt entsprechend der Zahlenbasis in **BASE** . Eine Fehlerbedingung besteht, wenn die Ziffern des Strings nicht in eine Zahl verwandelt werden können. Durch Angabe eines Präfix (siehe **NUMBER?**) kann die Basis für diese Zahl modifiziert werden.

dpl (-- addr) "decimal point location"

ist eine Variable, die die Stellung des Dezimalpunktes angibt.

Ein Beispiel der Umwandlung von Zeichen in Zahlen:

In FORTH wird die Eingabe von Zahlen oft mit der allgemeinen Texteingabe und über die Befehle zur Umwandlung von Strings in Zahlen realisiert. In der Literatur wird dazu oft diese Lösung mit **QUERY** angeboten:

```
: in#   ( <string> -- d tf  n tf  addr ff )
    query  bl word  number?  ;
```

Diese Lösung ist ungünstig, da QUERY den TIB löscht. Zugleich stellt die Definition von **NUMBER?** eine unglückliche Stelle im volksFORTH dar.
Es gibt im Laxen&Perry-F83 ein Wort gleichen Namens, das ganz anders (besser!) mit den Parametern umgeht. Hier folgt die Definition des F83-NUMBER?, aus dem volksFORTH **NUMBER?** aufgebaut:

```
: F83-NUMBER?  ( string -- d f )
   number?   ?dup IF  0< IF  extend  THEN  true exit  THEN
   drop 0 0 false ;
```

Damit stellt das Wort **INPUT#** eine wenig aufwendige Zahleneingabemöglichkeit für 16/32Bit-Zahlen dar:

```
\ input#                                        jrg 29jul88
: input#  ( <string> -- d f )
   pad  c/l 1-  >expect          \ get 63 char maximal
   pad  F83-number?   ;          \ converts string > number
```

So kann der Anwender das übergebene Flag auswerten und die doppelt-genaue Zahl entsprechend seinen Vorstellungen einsetzen, im einfachsten Fall mit **DROP** zu einer einfach-genauen Zahl machen.

6.4.2 Zahlen in Strings wandeln

(+dØ -- +d1) 83 "sharp"

Der Rest von +dØ geteilt durch den Wert in **BASE** wird in ein Ascii-Zeichen umgewandelt und dem Ausgabestring in Richtung absteigender Adressen hinzugefügt. +d1 ist der Quotient und verbleibt auf dem Stack zur weiteren Bearbeitung. Üblicherweise zwischen **<#** und **#>** benutzt.

#s (+d -- Ø Ø) 83 "sharp-s"

+d wird mit **#** umgewandelt, bis der Quotient zu Null geworden ist. Dabei wird jedes Zwischenergebnis in ein Ascii-Zeichen umgewandelt und dem String für die strukturierte Zahlenausgabe angefügt. Wenn +d von vornherein den Wert Null hatte, wird eine einzelne Null in den String gegeben. Wird üblichwerweise zwischen **<#** und **#>** benutzt.

hold (char --) 83

Das Zeichen char wird in den Ausgabestring für die Zahlenausgabe eingefügt. Wird üblicherweise zwischen **<#** und **#>** benutzt.

sign (n --) 83

Wenn n negativ ist, wird ein Minuszeichen in den Ausgabestring für die Zahlenausgabe eingefügt. Wird üblicherweise zwischen **<#** und **#>** benutzt.

#> (32b -- addr len) 83 "sharp-greater"

Am Ende der strukturierten Zahlenausgabe wird der 32b Wert vom Stack entfernt. Hinterlegt werden die Adresse des erzeugten Ausgabestrings und eine positive Zahl als die Anzahl der Zeichen im Ausgabestring, passend z.B. für **TYPE** .

7. Umgang mit Dateien

Das File-Interface wurde grundlegend überarbeitet.
Auf der Benutzerebene stehen die gleichen Worte wie im volksFORTH 3.8Ø für den ATARI und für CP/M zur Verfügung; die darunterliegende Implementation wurde jedoch grundlegend geändert, so daß jetzt in FORTH auch sequentielle Files, die nicht die starre BLOCK-Struktur haben, manipuliert werden können.

Damit ist es endlich möglich, auch volksFORTH für kleine Hilfsprogramme zu verwenden, die mit anderen Programmen erstellte Files "bearbeiten" und durch den Befehl **SAVESYSTEM** als "standalone"-Programm abgespreichert wurden.
Besonders weitreichende Möglichkeiten erschließen sich dadurch, daß beim Aufruf von volksFORTH auf der Betriebssystemebene noch eine ganze Kommandozeile mit übergeben werden kann, die dann unmittelbar nach dem Booten von FORTH ausgeführt wird. Durch die Systemvariable **RETURN_CODE** kann nach Verlassen des FORTHProgramms ein Wert an MS-DOS zurückgegeben werden, der mit dem Batch-Befehl ERRORLEVEL abgefragt werden kann.

Darüberhinaus ist es auch möglich, mit dem Befehl **MSDOS** aus dem FORTH heraus eine weitere COMMAND.COM shell aufzurufen und später mit EXIT wieder ins FORTH zurückzukehren, wobei der Bildschirm, der zum Zeitpunkt des Aufrufs bestand, wiederhergestellt wird.

Selbstverständlich kann neben MSDOS selber auch jedes andere beliebige Anwendungsprogramm aufgerufen werden - auch eine weitere Inkarnation des FORTH-Systems - so daß sich mit diesen Möglichkeiten die Begrenzungen überwinden lassen, die in dem beschränkten Adreßraum von 64k liegen. Auch komplizierte Overlaystrukturen sind nicht mehr notwendig, es werden einfach aus einem zentralen "Verwaltungsprogramm" heraus spezielle FORTH-Anwendungsprogramme aufgerufen.

Das Fileinterface des volksFORTH benutzt die Dateien des MSDOS und dessen Subdirectories.

Dateien bestehen aus einem FORTH-Namen und einem MSDOS-Namen, die nicht übereinstimmen müssen.
Ist das FORTH-Wort, unter dem ein File zugreifbar ist, gemeint, so wird im folgenden vom (logischen) FORTH-File gesprochen. Dies entspricht einer Dateivariablen in den PASCAL-ähnlichen Sprachen. Der Zugriff auf eine Datei erfolgt daher in volksFORTH über den Namen eines Files ; dieser Namen ermöglicht dann den Zugriff entweder auf den Datei-Steuerblock (FCB) oder die MSDOS-Handle-Nummer.
Ist das File auf der Diskette gemeint, das vom MSDOS verwaltet wird, so wird vom (physikalischen) DOS-File gesprochen.

Durch das Nennen des FORTH-Namens wird das FORTH-File (und das zugeordnete DOS-File) zum aktuellen File, auf das sich alle Operationen wie LIST , LOAD , CONVEY usw. beziehen.
Beim Öffnen eines Files wird die mit PATH angegebene Folge von Pfadnamen durchsucht. Dabei ist PATH ein von MSDOS völlig unabhängiger Pfad, so daß verschiedene Projekte auch mit verschiedenen FORTH-Systemen über ihren eigenen Pfad verwaltet werden können.
Der gültige MSDOS-Pfad wird davon nicht beeinflußt. Ist eine Datei einmal geöffnet, so kann diese Folge beliebig geändert werden, ohne daß der Zugriff auf das File behindert wird. Aus Sicherheitsgründen empfiehlt es sich aber, Files so oft und so lange wie irgend möglich geschlossen zu halten. Dann kann eine Änderung der Folge von Pfadnamen in PATH dazu führen, daß ein File nicht mehr gefunden wird.

volksFORTH bearbeitet seine Quelltexte in sogenannten Screen-Files, die üblicherweise die Endung .SCR haben.
Dies sind Files, die in 1 KB große Screens aufgeteilt sind, jeweils in 16 Zeilen zu je 64 Zeichen strukturiert.

Üblicherweise enthält der Screen Ø eines jeden Files eine kurze Erklärung über den Inhalt des Files - dies ist auch deshalb sinnvoll, da der Screen Ø eines Files nie geladen werden kann.

Screen 1 enthält üblicherweise den sogenannten "loadscreen". Dieser steuert den Ladevorgang des gesamten Files. Ein Beispiel für einen solchen LOADSCREEN ist der Scr1 der Datei VOLKS4TH.SYS, der die aktuelle Arbeitsumgebung VOLKS4TH.COM zusammenstellt.

In Zeile Ø eines jeden Screens ist ein Kommentar über den Inhalt des Screens und das Datum der letzten Änderung enthalten.

Um - vor allem auf Festplatten - Directories anzusprechen, gibt es verschiedene Methoden. Um beispielsweise Files in einem Directory "WORK" zu suchen oder anzulegen, geben Sie ein :

```
cd work
```

Jetzt werden alle Files und Sub-Directories im Directory WORK gesucht oder neu geschaffen. Ist das Directory WORK noch nicht vorhanden, so erzeugen Sie es mit :

```
md work
```

Wenn Sie Dateien auf einem bestimmten Laufwerk, z.B. A: , ansprechen wollen, geben Sie ein:

```
A:
```

Hierbei wird A: zum aktuellen Laufwerk gemacht.

Im einfachsten Fall existiert die zu benutzende Datei auf dem Massenspeicher und der Datei-Name bereits im volksFORTH-Dictionary. Solche Files sprechen Sie an, indem Sie einfach den Filenamen eingeben.

Mit dem Wort **files** zeigt Ihnen volksFORTH eine Liste dieser bereits im FORTH-System vorhandenen Dateien und Dateinamen, zusammen mit den zugeordneten DOS-Dateien und den Handle-Nummern.
Melden Sie irrtümlich eine Datei an, deren Name nicht im FORTH-Wörterbuch gefunden wird, erscheint die übliche Fehlermeldung, das "?".

Um dagegen zu kontrollieren, ob eine Datei auf Disk existiert, benutzen Sie **dir** .
Soll in einem solchen Fall ein File, das auf der Disk schon existiert, neu von volksFORTH bearbeitet werden, so wird es durch **use <filename>** zum sogenannten "aktuellen" File gemacht, in die Liste **FILES** und ins Wörterbuch eingetragen.
So macht **use test.scr** das MSDOS-File TEST.SCR zum aktuellen File, auf das sich solange alle weiteren File-Operationen beziehen, bis ein anderes File zum aktuellen File gemacht wird. Der Zugriff auf verschiedene Laufwerke ist für die Laufwerke A: bis H: vordefiniert.

Das Wort **use** erzeugt deshalb im FORTH-System das Wort TEST.SCR, falls es noch nicht vorhanden war. Wissen Sie also nicht mehr, ob Sie ein File schon benutzt haben, so können Sie das Wort **use** voranstellen, mit **files** nachsehen oder probehalber den Datei-Namen eingeben.

Möchten Sie ein neue Datei oder ein eigenes Screenfile anlegen, so wählen Sie einen Namen und benutzen:

makefile <filename>

Danach erfolgen die Schreib/Lesezugriffe des DOS auf <filename>. Bei einem Blockfile schätzen Sie die notwendige Größe nn in Blocks ab und geben ein:

makefile <filename> nn more (z.B. 20 more)

Dabei ist der Name maximal 8 Buchstaben und die Extension maximal 3 Buchstaben lang. Diese Extension kann, muß aber nicht angegeben werden. Als Konvention wird vorgeschlagen, daß Files, die FORTH-Screens, also Quelltexte, enthalten, die Endung .SCR erhalten. Files, die Daten enthalten, die nicht unmittelbar lesbar sind, sollten auf .BLK enden.

Auf **makefile** hin legt das System eine neue Quelltext-Datei von beispielsweise 20 KB Größe an. Das System geht bei 20KB davon aus, daß 10 KB Programmtexte in den ersten zehn Blöcken (0 - 9) und 10 KB Kommentare in den Shadow-Screens 10 - 19 eingetragen werden. Das File wird nur logisch (!) in der Mitte geteilt. Wenn auf allen Screens nur Quelltexte eingetragen werden, so hat das keine negativen Konsequenzen, Sie bekommen dann nur im Editor mit SHIFT-F9 keinen Kommentar angezeigt.

Sie haben nun eine Datei, das die Screens 0..19 enthält. Geben Sie jetzt ein:

1 l oder **1 edit**

Nach der Abfrage einer dreibuchstabigen Kennung, die Sie mit <CR> beantworten können, editieren Sie jetzt den Screen 1 Ihres neuen Files test.scr . Sie können, falls der Platz nicht ausreicht, Ihr File später einfach mit **more** verlängern. Ein

File kann leider nicht verkürzt werden.
Mit dem volksFORTH-Editor können Sie selbstverständlich auch stream files ohne die FORTH-typische Blockstruktur ansehen und editieren.

Drei wichtige Wörter für screen files sind:

Mit <nn> **list** wird Screen nn auf dem Bildschirm angezeigt - also zum Beispiel mit **1 list** der Screen 1 des Files TEST.SCR.

Mit <nn> **load** wird ein Screen nn geladen, d.h. durch den FORTH-Compiler in das Wörterbuch compiliert.

Mit **include** <filename> kann man unkompliziert ein ganzes Screenfile laden. Diese Operation ist der Sequenz **use** <filename> **1 load** äquivalent.

Beim Vergessen eines FORTH-Files mit Hilfe von **FORGET** , **EMPTY** usw. werden automatisch alle Blockpuffer, die aus diesem File stammen, gelöscht und, wenn sie geändert waren, auf die Diskette zurückgeschrieben. Das File wird anschließend geschlossen. Bei Verwendung von **flush** werden alle Files geschlossen. **FLUSH** sollte vor jedem Diskettenwechsel ausgeführt werden, und zwar nicht nur, um die geänderten Blöcke zurückzuschreiben, sondern auch damit alle Files geschlossen werden. Sind nämlich Files gleichen Namens auf der neuen Diskette vorhanden, so wird eine abweichende Länge des neuen Files vom FORTH nicht erkannt. Bei Verwendung von **VIEW** wird automatisch das richtige File geöffnet.

Sind bereits sehr viele Files geöffnet, kommt es in der MSDOS-Grundeinstellung recht schnell zu der Fehlermeldung des Betriebssystems: too many open files . Dagegen hilft die Änderung der Datei CONFIG.SYS:

```
FILES=16
```

Eine Besonderheit volksFORTH ist der Direktzugriff auf den Massenspeicher. Das Wort **DIRECT** schaltet das FileInterface ab und der Massenspeicher wird direkt physikalisch angesprochen !
Wenn nach dem Befehl **EMPTY** die Meldung F: direct auftaucht, ist das Fileinterface abgeschaltet. Solch ein Direktzugriff kann wieder auf das Fileinterface umgeschaltet werden, indem man einen Dateinamen aufruft. Bei den Dateinamen in der Liste der **FILES** ist also kein **USE** notwendig.
Wenn Sie als Beispiel direkt auf Drive A: zugreifen wollen, so ist die Syntax:

```
direct a:
```

Nun kann mit **list** , **load** , aber auch mit **edit** (!) auf die Disk zugegriffen werden! Deshalb ist bei Festplatten besondere Vorsicht geboten, da der Schreib-/Lesezeiger direkt auf dem Bootsektor der Platte stehen kann. Sollte Ihnen dies passieren, verlassen Sie den Editor mit CTRL-U (undo) und <ESC>.

Beim Direktzugriff muß allerdings beachtet werden, daß es für volksFORTH nur noch 1 KB große Blöcke gibt, die hintereinander liegen und nur durch ihre Blocknummern identifiziert sind. Eine 360K DS/DD-Disk enthält dann für volksFORTH keine Files und kein Directory, sondern es steht lediglich Platz für 362 Blöcke zur Verfügung.

Es gibt zwar einige Leute, die diese direct-Funktion dazu benutzen, um mit dem FORTH-Editor die Systembereiche ihrer Festplatte zu editieren; aber normalerweise ermöglicht der Direct-Zugriff dem volksFORTH, ohne MSDOS auszukommen und den Massenspeicher sehr schnell und effizient zu bedienen.

Einige wichtige Worte:

call (--)

ermöglicht den Aufruf von *.COM und *.EXE-Files.
call \park würde im Root-Directory Ihrer Harddisk ein Programm namens PARK aufrufen und ausführen.

msdos (--)

verläßt das volksFORTH zur MSDOS-Ebene; mit exit kehren Sie ins FORTH zurück.

savesystem <name> (--)

schreibt eine bootbare Form des jetzt laufenden FORTH-Systems unter dem Namen <name> auf den Massenspeicher. Dabei muß <name> die Endung .COM haben, wenn dieses System später wieder unter MSDOS gestartet werden soll.
Dieses Wort wird benutzt, um fertige Applikationen zu erzeugen. In diesem Fall sollte das Wort, daß die Applikation ausführt, in 'COLD gespeichert. Ein Beispiel:
Sie haben ein Kopierprogramm geschrieben, das oberste ausführende Wort heißt COPYDISK . Mit der Befehlsfolge

```
' copydisk Is 'cold
savesystem copy.com
```

erhalten Sie ein Progamm namens COPY.COM , das sofort beim Start COPYDISK ausführt.

direct (--)

schaltet das MSDOS-Fileinterface aus und auf Direktzugriff um. Auf den Filezugriff schalten Sie durch das Nennen eines Filenamens um.

7.1 Pfad-Unterstützung

volksFORTH besitzt eine eigene, vollständig von MSDOS unabhängige Pfadunterstützung. Die Verwaltung von Directories entspricht in etwa der des MSDOS-Command-interpreters COMMAND.COM .

Beispiel: `PATH A:\;B:\COPYALL\DEMO;\AUTO\`

In diesem Beispiel wird zunächst die Diskstation A, dann das Directory COPYALL\DEMO auf Diskstation B und schließlich das Directory AUTO auf dem aktuellen Laufwerk gesucht.
Beachten Sie bitte das Zeichen "\" vor und hinter AUTO . Das vordere Zeichen "\" steht für das Hauptdirectory. Wird es weggelassen, so wird AUTO\ an das mit DIR gewählte Directory angehängt. Das hintere Zeichen "\" trennt den Filenamen vom Pfadnamen.
Durch die vollständige Unabhängigkeit kann z.B. volksFORTH mit dem Pfad `C:\4TH;\4TH\WORK` arbeiten, während MSDOS den Pfad `C:\UTIL\TOOLS` absucht.

path (--)

Dieses Wort gestattet es, mehrere Directories nach einem zu öffnenden File durchsuchen zu lassen. Das ist dann praktisch, wenn man mit mehreren Files arbeitet, die sich in verschiedenen Directories oder Diskstationen befinden. Es wird in den folgenden Formen benutzt:

```
path dir1;\dir2\subdir2;\dir3
```

Hierbei sind <dir1>, <dir2> etc. Pfadnamen. Wird ein File auf dem Massenspeicher gesucht, so wird zunächst <dir1>, dann <dir2> etc. durchsucht. Zum Schluß wird das File dann im aktuellen Directory (siehe DIR) gesucht. Beachten Sie bitte, daß keine Leerzeichen in der Reihe der Pfadnamen auftreten dürfen.

path ohne Argument zeigt die aktuelle Pfadangabe und druckt die Reihe der Pfadnamen aus. Dieses Wort entspricht dem Kommando PATH des Kommandointerpreters.

path ; löscht die aktuelle Pfadangabe und es wird nur noch das aktuelle Directory durchsucht.

7.2 DOS-Befehle

dos

Viele Worte des Fileinterfaces, die normalerweise nicht benötigt werden, sind in diesem Vokabular enthalten.

dir

zeigt das Inhaltsverzeichnis des Massenspeichers. Ohne weitere Eingaben wird der aktuelle Suchpfad einschließlich des Laufwerks angezeigt.

delete <filename>

löscht eine Datei <filename> im aktuellen Directory .

ren <filename.alt> <filename.neu>

benennt eine Datei nach üblicher MSDOS-Syntax um.

ftype <filename>

gibt eine Text-Datei aus. **FTYPE** wurde so genannt, weil es ein FORTH-Wort **type** gibt.

md <dir>

legt ein neues Directory an.

cd <dir>

wechselt Directory (mit Pfadangabe).

rd <dir>

löscht ein Directory.

dos: FORTH-Name DOS-Name "

DOS: erwartet nach seinem Aufruf zwei Namen: Zuerst den FORTH-Namen, den der Befehl haben soll, dann den Namen der DOS-Funktion, die ausgeführt werden soll. Diese wird von einem Leerzeichen und einem quote " abgeschlossen. Ein Beispiel ist:

```
dos: ftype type "
```

Danach wird beim Aufruf des FORTH-Wortes **FTYPE** der MSDOS-Befehl **TYPE** ausgeführt.

7.3 Block-orientierte Dateien

Damit volksFORTH effizient auf den Massenspeicher zugreifen kann, bietet es - wie die meisten anderen FORTH-Systeme auch - einen besonderen Puffermechanismus für den Diskzugriff. Dadurch wird der Geschwindigkeitsunterschied zwischen den langsamen Laufwerken und dem schnellen Hauptspeicher ausgeglichen.
Entgegen der Literatur [Brodie] ist im volkFORTH das Ende eines Screens nicht durch ein Doppel-Nullbyte ØØ gekennzeichnet. volksFORTH arbeitet statt dessen mit **SKIP** und **SCAN** und wertet die Länge aus, die in **>in** bereits abgearbeitet ist.

view (<name> --)

wird in der Form **view <name>** benutzt.
Wenn <name> im Wörterbuch gefunden wird, so wird das File geöffnet, in dem der Quelltext von <name> steht und der Block wird gelistet, auf dem <name> definiert ist.
Siehe: **LIST**

list (blk# --)

zeigt den block mit der angegebenen Nummer an.

files (--)

zeigt alle FORTH-Dateien an. Diese müssen nicht mit den MSDOS-Files identisch sein. Dagegen zeigt **dir** die MSDOS-Files an.

isfile (-- addr)

ist die Adresse einer Variablen, die auf das aktuelle FORTH-File zeigt. Der Wert in dieser Variablen entspricht typisch der Adresse des File-ControlBlocks FCB. Ist der Wert in dieser Adresse gleich Null, so wird direkt ohne ein File auf den Massenspeicher zugegriffen.

isfile@ (-- fcb)

holt den FCB des isfile. Die Sequenz

```
isfile@ f.handle @
```

liefert die DOS-Handlenummer.

fromfile (-- addr)

ist die Adresse einer Variablen, die auf das FORTH-File zeigt, aus dem **copy** und **convey** die Blöcke lesen.
fromfile @ liefert den FCB dieser Datei.

fswap (--)

vertauscht **isfile** und **fromfile** .
Vergleiche im Editor das Drücken der Taste F9.

[FCB] (-- fcb)

ist eine Konstante des Typs FCB und dient für Vergleiche, ob ein Wort einen FCB darstellt. Es ist definiert als:

```
' kernel.scr @
```

.file (fcb --) DOS

druckt den FORTH-Filenamen des Files aus, dessen FCB-Adresse auf dem Stack liegt.

filename (-- addr) DOS

ist die Anfangsadresse eines 62-Byte großen Speicherbereichs, der zum Ablegen von Filenamen während DOS-File-Operationen dient.

fnamelen (-- n) DOS

ist eine Konstante, die die maximale Länge von logischen Filenamen, bestehend aus Drive, Path und Name, die in den FCB's abgespeichert werden können, bestimmt. Wird dieser Wert verändert, so kann die neue Länge erst in den FCB's verwendet werden, die nach der Änderung angelegt werden.

A: B: C: D: E: F: G: H: (--)

Entspricht MS-DOS, macht das dadurch bezeichnete logische Laufwerk zum aktuellen Laufwerk.

DTA (-- $80)

ist als Konstante die Start-Adresse der DiskTransferArea.

capacities (-- addr)

ist die Adresse eines Vektors, der die Kapazitäten der angeschlossenen logischen Laufwerken in 1kByte-Blöcken enthält. Dafür sind maximal 6 Einträge (siehe: **#DRIVES**) vorgesehen. Mit dem Hilfsprogramm DISKS.CFG können die Kapazitäten für die Diskettenlaufwerke eingestellt werden.

drv (-- #drive)

übergibt die Nummer des aktuellen Laufwerks.

drive (#drive --)

macht das Laufwerk mit der angegebenen Nummerzum aktuellen Laufwerk. Hierbei entspricht n=0 dem Laufwerk A, n=1 dem Laufwerk B usw. . Auf dem aktuellen Laufwerk werden Files und Directories erzeugt (und nach Durchsuchen von **PATH** gesucht).
0 block liefert die Adresse des ersten Blocks auf dem aktuellen Laufwerk.
Vergleiche **>drive** und **offset** .

file <name>

erzeugt ein FORTH-Wort mit dem gleichen Namen. Wird dieses Wort später aufgerufen, so vermerkt es sich als aktuelles File **isfile** und als Hintergund-Datei **fromfile** . Diesem logischen FORTH-File kann man dann mit **assign** oder **make** eine MSDOS-Datei zuordnen.

make <name>

wird in der Form

make <Dateiname>

benutzt. Es erzeugt ein MSDOS-File im aktuellen Directory und ordnet es dem aktuellen FORTH-File zu. Es hat die Länge Null (siehe **more**).
Beispiel: **file test.scr test.scr make test.scr**

makefile <name>

erzeugt eine FORTH-Datei mit diesem Namen und ebenfalls eine MSDOS-Datei mit dem gleichen Namen. Dieses File wird auch sofort geöffnet, hat aber noch die Länge Ø, wenn keine Größe mit **nn more** angegeben wurde. **makefile** ist ein Ersatz für die Prozedur in dem Beispiel, das bei **make** angegeben wurde.

assign (--)

wird benutzt in der Form

assign <filename>

und weist dem aktuellen logischen FORTH-File eine physikalische DOS-Datei zu.

file? (--)

zeigt den FORTH-Filenamen des aktuellen Files an.

?file (--)

zeigt den MSDOS-Namen, das Handle sowie Datum des letzten Zugriffs der aktuellen Datei.

load (blk# --)

lädt/compiliert den block mit der angegebenen Nummer an.

blk (-- addr)

ist die Adresse einer Variablen, die die Blocknummer des gerade interpretierten Blocks enthält. Wird **TIB** interpretiert, enthält **BLK** den Wert NULL ; d.h., auch wenn Sie den Inhalt interaktiv auslesen, so erhalten Sie immer den Wert Ø !

loadfrom (blk# --)

wird in folgender Form benutzt:

(block#) loadfrom <Dateiname>

Dies ist wichtig, wenn während des Compilierens Teile eines anderen Files geladen werden sollen. Damit kann die Funktion eines Linkers imitiert werden.

Beispiel: 15 loadfrom tools.scr

Dieses Wort benutzt use, um Files zu selektieren; das bedeutet, daß in diesem Beispiel automatisch eine Datei namens tools.scr erzeugt wird, falls dieses File noch nicht existierte.

Dieses Wort hat nichts mit **from** oder **fromfile** zu tun, obwohl es ähnlich heißt.

include (--)

wird in der Form benutzt:

include <Dateiname>

Hier wird ein File vollständig geladen. Voraussetzung ist, daß in Screen #1 Anweisungen stehen, die zum Laden aller Screens führen.

b/blk (-- &1024) "bytes pro block"

ist die Länge eines Blocks (bzw. Screens, immer 1 KByte) wird auf den Stack gelegt.

Siehe **B/BUF** .

b/buf (-- n) "bytes pro buffer"

n ist die Länge eines Blockpuffers incl. der Verwaltungs-informationen des Systems, in der Version 3.81 für den PC $408 Byte, bestehend aus $400 Byte Puffer und $8 Byte Verwaltungsinformation.

blk/drv (-- n) "blocks pro drive"

n ist die Anzahl der auf dem aktuellen Laufwerk verfügbaren Blöcke.

(more (n --)

wie **MORE** , jedoch wird das File nicht geschlossen.

more (n --)

vergrößert die aktuelle Datei (isfile) um eine bestimmte Anzahl von Blöcken, die hinter angehängt werden. Anschließend wird die Datei geschlossen.

capacity (-- n)

n ist die Speicherkapazität einer Datei, wobei Block Ø mitgezählt wird. Deshalb ist für Quelltexte nur capacity 1- (d.h. minus Block Ø) nutzbar. Wenn shadow screens verwendet werden, steht nur capacity 2/ 1- zur Verfügung.

block (u -- addr) 83

addr ist die Adresse des ersten Bytes des Blocks u in dessen Blockpuffer. Der Block u stammt aus dem File in **ISFILE** .

BLOCK prüft den Pufferbereich auf die Existenz des Blocks Nummer u. Befindet sich der Block u in keinem der Blockpuffer, so wird er vom Massenspeicher in einen an ihn vergebenen Blockpuffer geladen. Falls der Block in diesem Puffer als **UPDATE**d markiert ist, wird er auf den Massenspeicher gesichert, bevor der Blockpuffer an den Block u vergeben wird.

Nur die Daten im letzten Puffer, der über **BLOCK** oder **BUFFER** angesprochen wurde, sind sicher zugreifbar. Alle anderen Blockpuffer dürfen nicht mehr als gültig angenommen werden (möglicherweise existiert nur 1 Blockpuffer).

Vorsicht ist bei Benutzung des Multitaskers geboten, da eine andere Task **block** oder **buffer** ausführen kann. Der Inhalt eines Blockpuffers wird nur auf den Massenspeicher gesichert, wenn der Block mit **update** als verändert gekennzeichnet wurde.

(block (blk file -- addr)

liest den Block blk aus dem File, dessen FCB bei der Adresse file beginnt und legt diesen in einen Puffer bei der Adresse addr ab.

buffer (block# -- addr)

weist dem block mit der angegebenen Nummer einen Blockpuffer im Speicher zu. Die Zuweisung wird von der internen Logik vorgenommen - anschließend wird die Startadresse dieses Pufferbereiches übergeben. Dabei entspricht addr der Adresse des ersten Bytes des Blocks in seinem Puffer. Der große Unterschied zwischen **block** und **buffer** ist, daß **block** tatsächlich die Daten von Disk in den Puffer liest. **buffer** dagegen reserviert nur den Puffer, initialisiert ihn nicht und liest keine Daten vom Massenspeicher. Daher ist der Inhalt dieses Blockpuffers undefiniert.

(buffer (blk file -- addr)

reserviert einen 1kByte großen Puffer im Adreßbereich des FORTH-Systems für den Block blk. file ist die Adresse des FCB's, in dem sich der Block befindet. Ist file = Ø, dann handelt es sich um einen DIRECTen physikalischen Zugriff. addr ist die Anfangsadresse des Puffers.

offset (-- addr)
liefert die Adresse einer UserVariablen, deren Inhalt zu der Blocknummer addiert wird, die sich beim Aufruf von **block** , **buffer** usw. auf dem Stack befindet.

fblock@ (addr blk fcb --) DOS
1Ø24 Bytes, die im File fcb in Block blk stehen, werden ab der Adresse addr im FORTH-Adressbereich abgelegt.

fblock! (addr blk fcb --) DOS
1Ø24 Bytes, die ab der Adresse addr innerhalb des FORTH-Adressbereichs stehen, werden auf den Block blk innerhalb des Files geschrieben, das durch fcb characterisiert ist.

*block (blk -- d) DOS
Die doppeltgenaue Zahl d ist die Byteadresse des ersten Bytes im 1Ø24-Byte großen Block blk.

/block (d -- rest blk) DOS
Die doppeltgenaue Zahl D wird umgerechnet in die REST-Anzahl von Bytes innerhalb des 1Ø24-Byte großen Blocks blk.

r/w (addr blk fcb r/w -- *f)
ist ein deferred Wort, bei dessen Aufruf das systemabhängige Wort für den blockorientierten Massenspeicherzugriff ausgeführt wird.
Dabei ist addr die Anfangsadresse des Speicherbereiches für den Block block, fcb die Addresse des Files, in dem sich der Block befindet. r/w ist gleich Null, wenn vom Speicherbereich auf den Massenspeicher geschrieben werden soll oder r/w =1, wenn der Block gelesen werden soll. *f ist nur im Fehlerfall wahr, sonst falsch.

(r/w (addr blk fcb r/w -- *f)
Die Standardroutine für das deferred Wort R/W.

eof (-- true)
entspricht der Konstanten **TRUE** = -1 .

7.4 Verwaltung der Block-Puffer

core? (blk# filename -- addr | ff)

prüft, ob sich der Block mit der angegebenen Nummer aus der genannten Datei bereits in einem der Block-Puffer befindet. Wenn das der Fall ist, wird die Anfangsadresse des Puffers übergeben, anderenfalls ein false flag .
Vergleiche **BLOCK** , **BUFFER** und **ISFILE** .

(core? (blk# filename -- addr | false)

ist die Standardroutine für das Wort **CORE?** .

update (--)

markiert den zur Zeit gültigen Pufferspeicher als verändert. Von dieser update-Kennzeichnung wird dann die Sicherung der Daten abhängig gemacht, wenn dieser Blockpuffer für einen anderen Block benötigt oder wenn **SAVE-BUFFERS** ausgeführt wird.
Vergleiche **PREV** .

save-buffers (--)

Das Alias **sav** im Editor heißt normalerweise **save-buffers** und rettet alle als **UPDATE**d markierten Pufferbereiche auf Disk und löscht die **UPDATE**-Kennzeichnungen. Hierbei bleibt die bestehende Zuweisung blk -> buffer erhalten, die Blöcke werden jedoch nicht verändert und bleiben für weitere Zugriffe im Speicher erhalten.

flush (--)

schreibt alle als **UPDATE**d markierten Pufferbereiche auf Disk und löscht die **UPDATE**-Markierung. Allerdings wird jetzt die Zuweisung BLK -> BUFFER zerstört. Zugleich wird die Datei geschlossen und das Inhaltsverzeichnis aktualisiert.
Vergleiche **SAVE-BUFFERS** und **EMPTY-BUFFERS** .

emptybuf (buffer.addr --)

empty-buffers (--)

löscht den Inhalt aller Blockpuffer, ohne die Daten von als **UPDATE**d markierten Blockpuffern auf den Massenspeicher zurückzuschreiben. Es zerstört die Zuweisung BLK -> BUFFER und löscht die **UPDATE**-Markierung.

freebuffer (--)

Entfernt den Blockpuffer mit der niedrigsten Adresse aus der Liste der Blockpuffer und gibt den dadurch belegten Speicherbereich frei. **FIRST** wird entsprechend um **B/BUF** erhöht. Ist der Inhalt des Puffers als **UPDATE**d markiert, so wird er zuvor auf den Massenspeicher gesichert. Gibt es im System nur noch einen Blockpuffer, so geschieht nichts.
Vergleiche **ALLOTBUFFER** .

allotbuffer (--)

Fügt der Liste der Blockpuffer noch einen weiteren hinzu, falls oberhalb vom Ende des Retunstacks dafür noch Platz ist. **FIRST** wird entsprechend geändert.
Vergleiche **FREEBUFFER** und **ALL-BUFFERS** .

all-buffers (--)

Belegt den gesamten Speicherbereich von **LIMIT** abwärts bis zum oberen Ende des Returnstacks mit Blockpuffern.
Siehe **ALLOTBUFFER** .

first (-- addr)

ist eine Variable, die einen Zeiger auf den Blockpuffer mit der niedrigsten Adresse darstellt. So liefert Ihnen **FIRST @** die Startadresse des Blockpufferbereiches.
Vergleiche **ALLOTBUFFER** .

limit (-- addr)

ist im Gegensatz zu **FIRST** keine Variable, sondern liefert direkt die addr, unterhalb der sich die Blockpuffer befinden. Das letzte Byte des obersten Blockpuffers befindet sich in addr-1.
Vergleiche **ALL-BUFFERS** und **ALLOTBUFFER**.

Die Anzahl der Blockpuffer im aktuellen System läßt sich so ausrechnen:

```
: #buf  ( -- Anzahl )
    limit  first @ -   b/buf /  ;
```

prev (-- addr)

addr ist die Adresse einer Variablen, deren Wert der Anfang der Liste aller Blockpuffer ist. Der erste Blockpuffer in der Liste ist der zuletzt durch **BLOCK** oder **BUFFER** vergebene.

offset (-- addr)

addr ist die Adresse einer Uservariablen, deren Inhalt zu der Blocknummer addiert wird, die sich bei Aufruf von **BLOCK** , **BUFFER** usw. auf dem Stack befindet. **OFFSET** wird durch **DRIVE** verändert.

.status (--)

ist ein defered Wort, das vor dem Laden eines Blockes oder vor dem Abarbeiten der Tastatureingabe ausgeführt wird. Normalerweise ist es mit NOOP vorbesetzt.

7.5 Index-, Verschiebe- und Kopierfunktionen für Block-Files

Die meisten FORTH-Systeme, die bevorzugt mit Block-Dateien arbeiten, verfügen über eine INDEX-Funktion, die die Zeile Ø eines jeden Screens anzeigt. Diese Zeile gibt üblicherweise den Inhalt des Screens an.

Allgemein ist die Syntax für Index: <start> <end> index

Das nachfolgend beschriebene INDEX erwartet dagegen nur die Angabe des Blockes, bei dem die INDEX-Anzeige beginnen soll ; wird kein Argument übergeben, so beginnt die INDEX-Anzeige mit Screen 1. Auch können nach dem Kompilieren die durch ! als headerless gekennzeichneten Worte mit clear gelöscht werden.

```
! : range   ( from to -- to+1 from )
      2dup u> IF swap THEN  1+ swap ;

! : .fname    isfile@ [ DOS ] .file ;

: index     ( from -- )
    depth Ø= IF 1 THEN  capacity 1-  range
    ." von " .fname
    ( range) DO cr I 4 .r space
                I block  c/l -trailing  type
                stop? IF leave THEN
             LOOP ;
```

copy (u1 u2 --)

Der Block u1 wird in den Block u2 kopiert.
Innerhalb einer Datei wird ein bereits beschriebener Zielblock überschrieben, der alte Inhalt des Blocks u2 ist verloren.
Vergleiche auch CONVEY .
Ist ein FROMFILE angemeldet, so arbeitet COPY immer in Bezug auf das FROMFILE . Der Befehl 3 4 copy kopiert dann den Block# 3 aus der Hintergrunddatei FROMFILE in die aktuelle Datei ISFILE und nicht innerhalb des ISFILE !

from <name>

gibt an, aus welchem File bei Datei-Operationen herauskopiert werden soll. Vergleiche auch die Datei STREAM.SCR .

```
from abc.scr 1 10 copy
```

kopiert den Block 1 von File **FROMFILE** in den Block# 10 ins aktuelle **ISFILE** , wobei der Zielblock leer sein muß; soll nicht überschrieben, sondern eingefügt werden, wird

```
from abc.scr 1 10  1  convey
```

eingesetzt.

convey (1st.block last.block to.block --)

Verschiebt die Blöcke von 1st.block bis einschließlich last.block nach to.block. Die Bereiche dürfen sich überlappen. Eine Fehlerbehandlung wird eingeleitet, wenn last.block kleiner als 1st.block ist.
Die Blöcke werden aus dem File in **FROMFILE** ausgelesen und in das File in **ISFILE** geschrieben. **FROMFILE** und **ISFILE** dürfen gleich sein, Das Beispiel

```
4 6 5 convey
```

kopiert die Blöcke 4,5,6 nach 5,6,7. Der alte Inhalt des Blockes 7 ist verloren, der Inhalt der Blöcke 4 und 5 ist gleich.
CONVEY wird auch benutzt, wenn man gerne einen freien Block in ein Blockfile einfügen möchte:

(INSERT (start# --)

fügt an der angegebenen Zielposition einen Block ein.

```
: (insert  ( start# -- )
     dup 1+  capacity 1- swap  1 more  convey ;
```

INSERT (--)

fügt vor dem aktuellen Block einen Block ein.

```
: insert   ( -- )
    scr @ dup (insert
              block b/blk blank  update ;
```

7.6 FCB-orientierte Dateien

pushfile (--) C

wird in :-Definitionen benutzt, um den aktuellen Zustand der File-variablen **ISFILE** und **FROMFILE** nach dem Ende der Colon-Definition wiederherzustellen.
Vergleichen Sie bitte den Mechanismus der lokalen Variablen: PUSH

open (--)

öffnet das aktuelle File zur Bearbeitung; ist aber in den meisten Fällen überflüssig, weil Files beim Zugriff automatisch geöffnet werden.

emptyfile (--)

legt eine leere Datei zum aktuellen Dateinamen an. Wird z.B in **MAKE** benutzt.

killfile (--)

löscht ohne weitere Rückfrage des aktuelle MSDOS-File; das aktuelle FORTH-File wird dabei nicht gelöscht, denn das FORTH-File ist wie eine Dateivariable in anderen Sprachen zu betrachten und dementsprechend als Variable mit **forget ‹filename›** zu löschen.

close (--)

schließt das aktuelle File und aktualisiert das Inhaltsverzeichnis des Massenspeichers.

flush (--)

schließt alle geöffneten Dateien und aktualisiert das Inhaltsverzeichnis des Massenspeichers. Zugleich werden alle Zwischenpuffer auf die Disk zurückgeschrieben.

fopen (fcb --)

ist im volks4TH nicht vorhanden, weil die erwartete Funktion von **FRESET** erfüllt wird. Deshalb kann man dies so definieren:

```
' FRESET Alias FOPEN
```

freset (fcb --) DOS

fcb ist die Adresse eines FileControlBlocks. Das dadurch charakterisierte File wird "zurückgesetzt", d.h. das File wird geöffnet (wenn es noch nicht geöffnet war) und der Schreib/Lesezeiger wird auf den Anfang des Files gesetzt.

fclose (fcb --) DOS

Das File, dessen FCB-Adresse auf dem Stack liegt, wird geschlossen.

fgetc (fcb -- 8b ¦ eof) DOS

Aus dem File, dessen FCB-Adresse auf dem Stack liegt, wird das nächste Byte gelesen und der Schreib/Lesezeiger um eine Position weitergerückt. Wenn das letzte Byte bereits gelesen war, wird die End-Of-File-Markierung -1 zurückgegeben.

fputc (8b fcb --) DOS

Das Byte 8B wird an der aktuellen Position des Schreib/Lesezeigers in das File FCB geschrieben. Dabei wird der Zeiger um eine Position weitergerückt.

file@ (dfaddr fcb -- 8b | eof) DOS

Das Byte an der 32-bit Position dfaddr im File, daß durch fcb charakterisiert ist, wird gelesen. Liegt dfaddr jenseits des letzten Bytes im File, so wird -1 zurückgegeben. Nach erfolgreichem Lesen steht der Lese-/Schreibzeiger hinter dem gelesenen Byte.

file! (8b dfaddr fcb --) DOS

Das Byte 8B wird an die Position dfaddr des Files fcb geschrieben.

fseek (dfaddr fcb --) DOS

Der Schreib/Lesezeiger des Files, das durch fcb charakterisiert ist, wird auf die Position dfaddr gesetzt. Dabei ist dfaddr eine doppeltgenaue Zahl, so daß maximal Files von 4-GByte Größe verwaltet werden können.

savefile (addr len --)

wird in der Form:

```
savefile <name>
```

benutzt und schreibt die Anzahl von len Bytes ab der Adresse addr in das neu erzeugte File mit dem Namen <name>.

lfsave (seg:addr quan string --)

erzeugt ein File mit dem Namen, der als gecounteter String an der Adresse string abgelegt ist und schreibt die Anzahl quan Bytes ab der erweiterten Adresse seg:addr in dieses neue File.

lfputs (seg:addr quan fcb --)

quan Bytes ab der erweiterten Adresse seg:addr werden ab der aktuellen Position des Schreib/Lesezeigers in das File geschrieben, das durch fcb charakterisiert ist. Danach steht der Schreib/Lesezeiger hinter dem letzten geschriebenen Byte. Um z.B. den Bildschirmspeicher in seinem aktuellen Zustand in eine Datei zu schreiben, wird es in der Form benutzt:

```
... open  video@ 0  c/dis  isfile@  lfputs  close ...
```

lfgets (seg:addr quan fcb -- #read) DOS

siehe: 'READ . Lediglich wird statt der Handlenummer die Adresse des FCB's des gewünschten Files angegeben. Entsprechend dem Beispiel bei LFPUTS lassen sich mit diesem Wort Bildschirmmasken direkt in den Bildschirmspeicher laden.

asciz (-- asciz)
holt das nächste Wort im Quelltext in den Speicher und legt es als null-terminierten String bei der Adresse asciz ab.

>asciz (string addr -- asciz)
Mit diesem Operator wird der gecountete String an der Adresse string umgewandelt in einen nullterminierten String, der an der Adresse addr abgelegt wird. asciz ist die Adresse, an der der neue String liegt.

counted (asciz -- addr len)
wird benutzt, um die Länge eines mit einer Null terminierten Strings zu bestimmen. asciz ist die Anfangsadresse dieses Strings (MS-DOS verwaltet Strings so), addr und len sind die Stringparameter, die z.B. von **TYPE** verarbeitet werden würden.

loadfile (-- addr)
ist eine Variable als Pointer auf das File, das gerade geladen wird.

file-link (-- addr)
ist eine Variable, die den Anfang zur Verwaltung einer Liste der File-Control-Blöcke (FCB) enthält. Der Inhalt von **FILE-LINK** zeigt auf den Anfang des Parameterfeldes des zuletzt definierten FCB's - und an dieser Stelle steht dann die Adresse des davor definierten FCB's usw., so daß dadurch alle FCB's aufgefunden werden können. Diese Liste wird u.a. von **forget** und **files** benutzt.

7.7 HANDLE-orientierte Dateien

~creat (asciz attribut -- handle ff ¦ err#) DOS
Der MS-DOS Systemaufruf, um ein neues File zu erzeugen.

~open (asciz mode -- handle ff ¦ err#) DOS
Der MS-DOS Systemaufruf für das Öffnen eines Files. asciz ist die Adresse des vollen Namensstrings und mode bezeichnet die Art des File-Attributes. Dabei sind mögliche Attribute :

```
0 Constant read-only
1 Constant write-only
2 Constant read-write
```

Bei Erfolg liegt eine HANDLE-Nummer unter einer Null auf dem Stack, ansonsten eine Fehlernummer.

~read (seg:addr quan handle -- #read) DOS
quan Bytes werden aus dem File gelesen, daß durch die Kennummer handle charakterisiert ist. Sie werden im erweiterten Speicherbereich bei seg:addr abgelegt. Nach Ende der Leseoperation liegt die Anzahl der Bytes auf dem Stack, die tatsächlich bis zum Ende des Files gelesen werden konnten. Es können jedoch nur maximal 64kByte auf einmal gelesen werden.

~close (handle --) DOS
Der MS-DOS Systemaufruf, um das File, das durch handle characterisiert ist, zu schließen.

~unlink (asciz -- err#) DOS
Der MS-DOS Systemaufruf, um einen Fileeintrag zu löschen.

~dir (addr drive -- err#) DOS
Der MS-DOS Systemaufruf, mit dem das aktuelle Directory an der Adresse addr als nullterminierter String abgelegt wird.

~select (n --) DOS
Der MS-DOS Systemaufruf, mit dem das aktuelle Laufwerk selektiert wird.

~disk? (-- n) DOS
Der MS-DOS Systemaufruf, mit dem das aktuelle Laufwerk abgefragt wird.

attribut (-- addr) DOS
Eine Variable, die die File-Attribute enthält, die bei der Suche nach Files in einem Directory berücksichtigt werden. Standardmäßig mit 7 initialisiert, so daß in die Suche read-only, hidden und system -Files eingeschlossen sind.

(fsearch (string -- asciz *f) DOS
Das File, dessen Name als String ab Adresse string steht, wird in der Directory gesucht. Enthält der Filename keine Suchpfadinformation, dann wird im aktuellen Directory gesucht. Bei Erfolg liegt eine Null auf dem Stack, sonst eine Fehlernummer.

fsearch (string -- asciz *f) DOS
Ein deferred Wort. Es enthält die Suchstrategie (siehe: (FSEARCH), die beim Öffnen eines Files verwendet wird, um das File auf der Disk zu lokalisieren.

~first (asciz attr -- err#) DOS
Der MS-DOS Systemaufruf, um erstmalig nach einem File zu suchen.

~next (-- err#) DOS
Der MS-DOS Systemaufruf, der nach ~FIRST benutzt wird, um weitere passende Filenamen aufzufinden.

7.8 Direkt-Zugriff auf Disketten

direct (--)
Die Filevariablen werden auf Null gesetzt und damit beziehen sich die Diskzugriffe durch BLOCK auf physikalische Blocks.

#drives (-- n)
ist eine Konstante, die die mögliche Anzahl von logischen Laufwerken im System definiert. Diese Anzahl ist nur im DIRECT-Modus von Bedeutung. So, wie der Kern compiliert ist, sind maximal 6 Laufwerke zugelassen.

/drive (blk1 -- blk2 drive)
für den DIRECT-Modus beim Diskzugriff. Aus der absoluten Blocknummer blk1 wird (siehe: CAPACITIES) die relative Blocknummer blk2 auf Laufwerk DRIVE berechnet. Dabei entspricht Laufwerk A: dem Drive Ø etc. .

>drive (blk1 #drv -- blk2) "to-drive"
dient zum "Umrechnen" von Blocknummern im DIRECT-Modus.
blk2 ist die absolute Blocknummer, die dem relativen Block blk1 auf Drive #drv entspricht. Beispiel

```
23 1 >drive block
```

holt den Block mit der Nummer 21 vom Laufwerk 1, egal welches Laufwerk gerade das aktuelle ist.

capacity (-- n)
gibt die Kapazität in 1Ø24-Byte Blöcken des aktuellen Files bzw. des aktuellen Laufwerks bei DIRECT-Zugriff.

7.9 Fehlerbehandlung

?diskerror (--)
Ein deferred Wort, daß die Fehlerbehandlungsroutine für Disk- und File-zugriffe enthält. Standardmäßig ist die Routine (DISKERROR zugewiesen.

error# (-- addr)

Eine Variable, die die Fehlernummer des letzten Fehlers beim Zugriff auf ein File enthält.

(diskerror (#err --) DOS

Die Standard-System Fehlerbehandlungsroutine für Fehler beim Diskzugriff. Hiermit ist das deferred Wort ?DISKERROR initialisiert.

8. Speicheroperationen

Die INTEL-Prozessoren haben eine verkomplizierte Art, den Adreßraum jenseits von 64kBytes zu adressieren - nämlich mit sogenannten "Segmentregistern". Am besten kommt man damit noch zurecht, wenn man diese Prozessoren als 16-bit Prozessoren betrachtet, die in der Lage sind, mehrere Programme, die jeweils höchstens 64k Programmspeicherbereich haben, gleichzeitig im Speicher zu halten. Es ist deshalb auch unvernünftig, auf diesen Prozessoren ein FORTH-System mit 32-bit Adressen zu installieren - es handelt sich eben nicht um 32-bit Prozessoren.

Um in volksFORTH den gesamten 1 MB-Adreßraum zu nutzen, ist die Möglichkeit gegeben, aus dem FORTH heraus mit dem Wort call (im DOS-Vokabular) ein weiteres .COM- oder .EXE-Programm aufzurufen. Dies können natürlich ihrerseits FORTH-Programme sein, denen dann auch noch eine ganze Eingabezeile als Parameter mit "auf den Weg" gegeben werden kann.
Damit wäre es zum Beispiel möglich, den Full-Screen Editor aus dem System auszulagern und mit den Befehlen **FIX** , **EDIT** , **ED** usw. jeweils ein .COM-Programm aufzurufen, das den FORTH-Editor als "stand-alone" Programm enthält und damit keinen Adreßraum im Entwicklungssystem mehr verbraucht.

Über die Systemvariable **RETURN_CODE** ist es auch noch möglich, einen Fehlercode bei Beendigung des FORTH-Programms an MS-DOS zu übergeben, der dann in Batch-Files getestet werden kann.

Die Intel-Prozessoren setzen die Speicheradressen aus zwei Teilen zusammen, dem SEGMENT und dem OFFSET. Dies ist jedoch nicht mit "echten" 32Bit-Adressen zu verwechseln. Diese werden auf einem 16Bit-Stack in der Reihenfolge "low-word" unter dem "high-word" abgelegt. Überträgt man diese Philosophie auf die "seg:addr"-Adressen des 8Ø86, dann blockiert dauernd die Segmentadresse den Stack.
Deshalb wird bei den Operatoren, die im erweiterten Adreßraum des 8Ø86 operieren, die Segmentadresse UNTER der Offsetadresse auf den Stack gelegt. Der Stackkommentar dafür lautet "seg:addr". Den Operatoren, die als Adreßargument eine "erweiterte" Adresse benötigen, wird ein "l" im Namen vorangestellt.

8.1 Speicheroperationen im 16-Bit-Adressraum

@ (addr -- 16b) 83 "fetch"

Von der Adresse addr wird der Wert 16b aus dem Speicher geholt. Siehe auch **!** .
volksFORTH enthält nicht das Wort **?** . Benutzen Sie:

```
: ?  ( addr --) @ . ;
```

! (16b addr --) 83 "store"
16b werden in den Speicher auf die Adresse addr geschrieben. In 8Bit-weise adressierten Speichern werden die zwei Bytes addr und addr+1 überschrieben.

+! (w1 addr --) 83 "plus-store"
w1 wird zu dem Wert w in der Adresse addr addiert. Benutzt die plus-Operation. Die Summe wird in den Speicher in die Adresse addr geschrieben. Der alte Speicherinhalt wird überschrieben.

2! (32b addr --) 83 "two-store"
32b werden in den Speicher ab Adresse addr geschrieben.

2@ (addr -- 32b) 83 "two-fetch"
Von der Adresse addr wird der Wert 32b aus dem Speicher geholt.

c! (16b addr --) 83 "c-store"
Von 16b werden die niederwertigsten 8 Bit in den Speicher an der Adresse addr geschrieben.

c@ (addr -- 8b) 83 "c-fetch"
Von der Adresse addr wird der Wert 8b aus dem Speicher geholt.

move (addrØ addr1 u --)
Beginnend bei addrØ werden u Bytes nach addr1 kopiert. Dabei ist es ohne Bedeutung, ob überlappende Speicherbereiche aufwärts oder abwärts kopiert werden, weil **MOVE** die passende Routine dazu auswählt. Hat u den Wert Null, passiert nichts.
Siehe auch **CMOVE** und **CMOVE>** .

cmove (addrØ addr1 u --) 83 "c-move"
Beginnend bei addrØ werden u Bytes zur Adresse addr1 kopiert. Zuerst wird das Byte von addrØ nach addr1 bewegt und dann aufsteigend fortgefahren. Wenn u Null ist, wird nichts kopiert.

cmove> (addrØ addr1 u --) 83 "c-move-up"
Beginnend bei adDrØ werden u Bytes zur Adresse adDr1 kopiert. Zuerst wird das Byte von addrØ +u -1 nach addr1 +u -1 kopiert und dann absteigend fortgefahren. Wenn u Null ist, wird nichts kopiert. Das Wort wird benutzt, um Speicherinhalte auf höhere Adressen zu verschieben, wenn die Speicherbereiche sich überlappen.

fill (addr u 8b --)

Von der Adresse addr an werden u Bytes des Speichers mit 8b überschrieben. Hat u den Wert Null, passiert nichts.

erase (addr u --)

Von der Adresse addr an werden u Bytes im Speicher mit $ØØ überschrieben. Hat u den Wert Null, passiert nichts.

blank (addr u --)

Von der Adresse addr an werden u Bytes im Speicher mit Leerzeichen BL ($2Ø) überschrieben. Hat u den Wert Null, passiert nichts.

flip (u1 -- u2)

ist das Byteswap des obersten Stackelements.
u1 ist eine 16-bit Zahl mit den Bits B15..B8 und B7..BØ, wobei man B15..B8 als das "high-Byte", B8..BØ als das "low-Byte" bezeichnet. Durch **FLIP** wird das High- mit dem Low-Byte ausgetauscht, so daß u2 als Ergebnis die Bits in der Reihenfolge B7..BØ und B15..B8 angeordnet hat. Dieses **FLIP** entspricht der Definition:

```
: cswap   ( 16b -- 16b' )  $100 um* or ;
```

ctoggle (8b addr --) 83 "c-toggle"

Für jedes gesetzte Bit in 8b wird im Byte mit der Adresse addr das entsprechende Bit invertiert (d.h. ein zuvor gesetztes Bit ist danach gelöscht und ein gelöschtes Bit ist danach gesetzt). Für alle gelöschten Bits in 8b bleiben die entsprechenden Bits im Byte mit der Adresse addr unverändert. Der Ausdruck **dup c@ rot xor swap c!** wirkt genauso.

off (addr --)

schreibt den Wert **FALSE** in den Speicher mit der Adresse addr.

on (addr --)

schreibt den Wert **TRUE** in den Speicher mit der Adresse addr.

here (-- addr) 83

addr ist die Adresse des nächsten freien Dictionaryplatzes.

align (--)

rundet normalerweise den Dictionary-Pointer und damit auch **HERE** auf die nächste geraden Adresse auf. Ist **HERE** gerade, so geschieht nichts. Im volks4th für den 8Ø88-Prozessor hat **ALIGN** keine Wirkung.

, (16b --) 83 "comma"
ist ein 2 ALLOT für die gegebenen 16b und speichert diese 16b an HERE 2- ab.

c, (8b --) "c-comma"
ist ein ALLOT für ein Byte und speichert 1 Byte in HERE 1- ab.

allot (w --) 83
allokiert w Bytes im Dictionary. Die Adresse des nächsten freien Dictionaryplatzes wird entsprechend verstellt.

dp (-- addr) "d-p"
ist eine Uservariable, in der die Adresse des nächsten freien Dictionaryplatzes steht.

uallot (n1 -- n2)
allokiert bzw. deallokiert n1 Bytes in der Userarea. n2 gibt den Anfang des allokierten Bereiches relativ zum Beginn der Userarea an. Eine Fehlerbehandlung wird eingeleitet, wenn die Userarea voll ist.

udp (-- addr) "u-d-p"
ist eine Uservariable, in dem das Ende der bisher allokierten Userarea vermerkt ist.

pad (-- addr) 83
addr ist die Startadresse einer "scratch area". In diesem Speicherbereich können Daten für Zwischenrechnungen abgelegt werden. Wenn die nächste verfügbare Stelle für das Dictionary verändert wird, ändert sich auch die Startadresse von PAD . Die vorherige Startadresse von PAD geht ebenso wie die Daten dort verloren.

count (addr1 -- addr2 8b) 83
addr2 ist addr1+1 und 8b der Inhalt von addr1. Das Byte mit der Adresse addr1 enthält die Länge des Strings angegeben in Bytes. Die Zeichen des Strings beginnen bei addr1+1. Die Länge eines Strings darf im Bereich (Ø..255) liegen.
Vergleiche "counted String" .

place (addr1 +n addr2 --) .
bewegt +n Bytes von der Adresse addr1 zur Adresse addr2+1 und schreibt den Wert +n in die Speicherstelle mit der Adresse addr2. Wird in der Regel benutzt, um Text einer bestimmten Länge als "counted string" abzuspeichern. addr2 darf gleich, größer und auch kleiner als addr1 sein.

dump (addr Anzahl --)

Dieser wichtige Befehl zeigt ab einer gegebenen Adresse eine bestimmte Anzahl Bytes des Hauptspeichers an. Denn ist es sinnvoll, nachdem man im Speicher ganze Bereiche reserviert und dort Strukturen abgelegt hat, sich diese Bereiche auch anzusehen. Ebenso zeigt Ihnen der Befehl

(n) block b/blk dump

einen Ihrer Quelltextblöcke an.

b/seg (-- n)

ist eine Konstante, die angibt, wieviele Bytes zwischen zwei Segmenten liegen. Dies sind beim 8Ø86 16 Bytes, beim 8Ø286 im 286-Modus jedoch 64 Bytes. volksFORTH auf dem 8Ø286 setzt zur Zeit voraus, daß der 8Ø86-Emulationsmodus eingeschaltet ist.

ds@ (-- seg)

legt die Segmentadresse des Segments auf den Stack, in dem sich das maximal 64kByte große FORTH-System gerade befindet. Das Daten-, Extra-, Stack- und Codesegment werden durch FORTH alle auf den gleichen Wert gesetzt.

8.2 Segmentierte Speicheroperationen

l@ (seg:addr -- n)

entspricht dem @ , jedoch im erweiterten Adreßraum.

l! (n seg:addr --)

entspricht dem ! , jedoch im erweiterten Adreßraum.

lc@ (seg:addr -- 8b)

entspricht dem C@ , jedoch im erweiterten Adreßraum.

lc! (8b seg:addr --)

entspricht dem C! , jedoch im erweiterten Adreßraum.

lmove (from.seg:addr to.seg:addr quan --)

entspricht dem MOVE , jedoch im erweiterten Adreßraum. Es können hiermit maximal 64KBytes auf einmal bewegt werden.

ltype (seg:addr len --)

entspricht dem **TYPE** , jedoch im erweiterten Adreßraum. Es ist zu beachten, daß **TYPE** in den Videodisplaytreibern BIOS.VID und MULTI.VID so implementiert ist, daß bei Erreichen des Zeilenendes nicht automatisch ein CR ausgeführt wird. Statt dessen werden alle Zeichen, die "jenseits" des rechten Rands liegen, nicht ausgegeben.

ldump (seg:addr quan --) TOOLS

entspricht dem **DUMP** , jedoch im erweiterten Adreßraum und zeigt ab einer angegebenen Adresse eine bestimmte Anzahl Bytes an.

lallocate (#pages -- seg ff | rest err#) EXTEND

Hiermit können im erweiterten Adreßraum die Anzahl #pages Speicherplatz angefordert werden.
Die Größe einer "Page" in Bytes entspricht der Konstanten **B/SEG** . Wenn die Speicheranforderung erfüllt werden kann, dann wird unter einer NULL als Flag für den Erfolg der Operation die Segmentadresse des ersten Segments innerhalb eines zusammenhängenden Speicherbereichs von #page Pages auf den Stack gelegt.
Ansonsten liegt unter einem Fehlercode die maximale Anzahl von Pages, die noch als zusammenhängender Bereich verfügbar sind. Diese Funktion ist in dem Wort **SAVEVIDEO** benutzt, um den Bildschirminhalt in den Speicher zu kopieren.
Die komplementären Funktionen sind **LFREE** und **RESTOREVIDEO** .

lfree (seg -- err#)

Der Speicherbereich, der an der Segmentadresse seg beginnt, wird wieder an das Betriebssystem zurückgegeben.
Diese Operation ist nur definiert, wenn zu einem vorherigen Zeitpunkt eine **LALLOCATE**-Operation durchgeführt worden war, die als Ergebnis die Segmentadresse seg gehabt hatte.
Es wird ein Fehlercode auf dem Stack übergeben, der im Erfolgsfall Ø ist.

9. Datentypen

Allgemein bestehen Programme aus einem Programm:Datenteil und einem Anweisungsteil. Dabei enthält der Befehlsteil die zu Algorithmen angeordneten Befehle, die sich wiederum aus Operanden und Operatoren zusammensetzen. Der Datenteil dagegen beschreibt eines Programmes den statischen Datenbereich, der die zu bearbeitenden Daten enthält.

Diese Datenbereiche sind durch die Interpretation ihres Inhaltes strukturiert, wobei Datenbereiche mit gleicher Strukturierung zum gleichen Datentyp gehören. Denn um mit einer Reihe von Bytes arbeiten zu können, muß man wissen, ob diese Bytes eine Zeichenkette, den Inhalt einer Variablen oder die Listenzellen einer :-Definition darstellen.

Die Strukturierung und Verwaltung sowie die Interpretation des Dateninhaltes im Datenteil wird in den PASCAL-ähnlichen Sprachen am Programmanfang durch die Deklarationen der Datentypen, in FORTH dagegen erst im Befehlsteil eines Programmes durch die eingesetzten Befehle bestimmt.

Da in FORTH die Typisierung über die Auswahl und den Einsatz von geeigneten Operatoren erfolgt, gehören in FORTH alle die Datenstrukturen dem gleichen Typ an, denen die Operatoren gemeinsam sind. In FORTH bestimmt also die jeweilige Operation den Datentyp des entsprechenden Bereiche - so interpretiert die Befehlsfolge pad count type die im Speicherbereich PAD abgelegten Daten als Zeichenkette !

Glossar

Create (--) 83

ist ein definierendes Wort, das in der Form:

Create <name>

benutzt wird. **CREATE** erzeugt einen Kopf für <name>.
Die nächste freie Stelle im Dictionary (vergl. **HERE** und **DP**) ist nach einem **CREATE <name>** das erste Byte des Parameterfelds von <name>. Wenn <name> ausgeführt wird, legt es die Adresse seines Parameterfelds auf den Stack. **CREATE** reserviert keinen Speicherplatz im Parameterfeld von <name>.

Constant (16b --) 83

ist ein definierendes Wort, das in der Form:

16b Constant <name>

benutzt wird. Wird <name> später ausgeführt, so wird 16b auf den Stack gelegt.

2Constant (32b --) 83

ist ein definierendes Wort, das in der Form:

32b 2Constant <name>

benutzt wird. Erzeugt einen Kopf für <name> und legt 32b in dessen Parameterfeld so ab, daß bei Ausführung von <name> 32b wieder auf den Stack gelegt wird.

Variable (--) 83

ist ein definierendes Wort, benutzt in der Form:

Variable <name>

VARIABLE erzeugt einen Kopf für <name> im Dictionary und hält 2 Byte in seinem Parameterfeld frei. Siehe **ALLOT** .
Dieses Parameterfeld wird für den Inhalt der Variablen benutzt, jedoch nicht initialisiert. Wird <name> ausgeführt, so wird die Adresse des Parameterfeldes von <name> auf den Stack gelegt. Mit @ und ! kann der Wert von <name> gelesen und geschrieben werden.
Siehe auch +! .

2Variable (--) 83

ist ein definierendes Wort, das in der Form:

2Variable <name>

benutzt wird. Es erzeugt einen Kopf für <name> und hält 4 Byte in seinem Parameterfeld frei, die den Inhalt der doppelt-genauen Variablen aufnehmen. **2VARIABLE** wird nicht initialisiert. Wenn <name> ausgeführt wird, wird die Adresse des Parameterfelds auf den Stack gelegt.
Siehe **VARIABLE** .

: (-- sys) 83 "colon"

ist ebenfalls ein definierendes Wort, das in der Form:

: <name> <actions> ;

benutzt wird.
Es erzeugt die Wortdefinition für <name> im Kompilations-Vokabular, schaltet den Kompiler an und kompiliert anschließend den Quelltext wird.

Achtung:
Das erste Vokabular der Suchreihenfolge - das "transient" Vokabular - wird durch das Kompilations-Vokabular ersetzt! Das Kompilations-Vokabular wird nicht geändert. .
Soll also das **CONTEXT**-Vokabular für eine :-Definition geändert werden, so ist das gewünschte Vokabular entweder innerhalb einer Definition mit den eckigen Klammern

```
: <name>   [ <vocabulary> ] ... ;
```

zum **CONTEXT** zu machen oder außerhalb einer Definition zweimal in die Suchreihenfolge aufzunehmen (**also**) und später zu löschen (**toss**) :

```
<vocabulary>    also
: <colondefinition> ;    toss
```

<name> wird als "colon-definition" oder ":-Definition" bezeichnet. Die neue Wortdefinition für <name> kann nicht im Dictionary gefunden werden, bis das zugehörige ; oder **;CODE** erfolgreich ausgeführt wurde. **RECURSIVE** macht <name> jedoch sofort auffindbar.
Vergleiche **HIDE** und **REVEAL** .

Eine Fehlerbehandlung wird eingeleitet, wenn ein Wort während der Kompilation nicht gefunden bzw. nicht in eine Zahl (siehe auch **BASE**) gewandelt werden kann.
Der auf dem Stack hinterlassene Wert sys dient der Kompiler-Sicherheit und wird durch ; bzw. **;CODE** abgebaut.

; (--) 83 I C "semi-colon"
(sys --) compiling

beendet die Kompilation einer :-Definition.
; macht den Namen dieser colon definition im Dictionary auffindbar, schaltet den Kompiler aus, den Interpreter ein und kompiliert ein **UNNEST** (siehe auch **EXIT**). Der Stackparameter sys, der in der Regel von : hinterlassen wurde, wird geprüft und abgebaut. Eine Fehlerbehandlung wird eingeleitet, wenn sys nicht dem erwarteten Wert entspricht.

Defer (--)

ist ein definierendes Wort, das in der Form:

```
Defer <name>
```

benutzt wird. Erzeugt den Kopf für ein neues Wort <name> im Dictionary, hält 2 Byte in dessen Parameterfeld frei und speichert dort zunächst die Kompilationsadresse einer Fehlerroutine. Wird <name> nun ausgeführt, so wird eine Fehlerbehandlung eingeleitet.

Man kann dem Wort <name> jedoch zu jeder Zeit eine andere Funktion zuweisen mit der Sequenz:

```
' <action> Is <name>
```

Nach dieser Zuweisung verhält sich <name> wie <action>.
Mit diesem Mechanismus kann man zwei Probleme elegant lösen:

1. Einerseits läßt sich <name> bereits kompilieren, bevor ihm eine sinnvolle Aktion zugewiesen wurde.
2. Andererseits ist die Veränderung des Verhaltens von <name> für spezielle Zwecke auch nachträglich möglich, ohne neu kompilieren zu müssen.

Deferred Worte im System sind **R/W** **'COLD** **'RESTART** **'ABORT** **'QUIT** **NOTFOUND** **.STATUS** und **DISKERR** . Diese Worte sind DEFERred, damit ihr Verhalten für die Anwendung geändert werden kann.

Is (cfa --)

ist ein Wort, mit dem das Verhalten eines deferred Wortes verändert werden kann. **IS** wird in der Form:

```
' <action> Is <name>
```

benutzt. Wenn <name> kein deferred Wort ist, wird eine Fehlerbehandlung eingeleitet, sonst verhält sich <name> anschließend wie <action>.
Siehe **DEFER** .

Alias (cfa --)

ist ein definierendes Wort, das typisch in der Form:

```
' <oldname> Alias <newname>
```

benutzt wird. **ALIAS** erzeugt einen Kopf für <newname> im Dictionary. Wird <newname> aufgerufen, so verhält es sich wie <oldname>. Insbesondere wird beim Kompilieren nicht <newname>, sondern <oldname> im Dictionary eingetragen.

Vocabulary (--) 83

ist ein definierendes Wort, das in der Form:

```
Vocabulary <name>
```

benutzt wird. **VOCABULARY** erzeugt einen Kopf für <name>, das den Anfang einer neuen Liste von Worten bildet. Wird <name> ausgeführt, so werden bei der Suche im Dictionary zuerst die Worte in der Liste von <name> berücksichtigt. Wird das **VOCABULARY** <name> durch die Sequenz:

```
<name> definitions
```

zum Kompilations-Vokabular, so werden neue Wort-Definitionen in die Liste von <name> gehängt.
Vergleiche auch **CONTEXT CURRENT ALSO TOSS ONLY FORTH** und **ONLYFORTH** .

Input: (--) "input-colon"

ein definierendes Wort, benutzt in der Form:

```
Input: <name>
  newKEY newKEY? newDECODE newEXPECT ;
```

INPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen Satz von Zeigern auf Worte, die für die Eingabe von Zeichen zuständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld von <name> in die Uservariable **INPUT** geschrieben. Alle Eingaben werden jetzt über die neuen Eingabeworte abgewickelt.

Die Reihenfolge der Worte nach **INPUT:** <name> bis zum semi-colon muß eingehalten werden: **..key ..key? ..decode ..expect** .

Im System ist das mit **INPUT:** definierte Wort **keyboard** enthalten, nach dessen Ausführung alle Eingaben von der Tastatur geholt werden.
Siehe **DECODE** und **EXPECT** .

Output: (--) "output-colon"

ist ein definierendes Wort, benutzt in der Form:

```
Output: <name>
   newEMIT newCR newTYPE newDEL newPAGE  newAT newAT? ;
```

OUTPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen Satz von Zeigern auf Worte, die für die Ausgabe von Zeichen zuständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld von <name> in die Uservariable **OUTPUT** geschrieben. Alle Ausgaben werden jetzt über die neuen Ausgabeworte abgewickelt.

Die Reihenfolge der Worte nach OUTPUT: <name> bis zum semi-colon muß eingehalten werden: **..emit ..cr ..type ..del ..page ..at ..at?** .

Im System ist das mit **OUTPUT:** definierte Wort **display** enthalten, nach dessen Ausführung alle Ausgaben auf den Bildschirm geleitet werden. Vergleiche auch das Wort **PRINT** aus dem Printer-Interface.

User (--) 83

ist ein definierendes Wort, benutzt in der Form:

```
User <name>
```

USER erzeugt einen Kopf für <name> im Dictionary und hält 2 Byte in der Userarea frei (siehe **UALLOT**). Diese 2 Byte werden für den Inhalt der Uservariablen benutzt und werden nicht initialisiert. Im Parameterfeld der Uservariablen im Dictionary wird nur ein Offset zum Beginn der Userarea abgelegt.
Wird <name> ausgeführt, so wird die Adresse des Wertes der Uservariablen in der Userarea auf den Stack gelegt.
Uservariablen werden statt normaler Variablen z.B. dann benutzt, wenn der Einsatz des Multitaskers geplant ist und mindestens eine Task die Variable unbeeinflußt von anderen Tasks benötigt. Jede Task hat ihre eigene Userarea.

9.1 Ein- und zweidimensionale Felder

Wie in dem Buch "In FORTH denken" [S.192/193] diskutiert wird, stellen die meisten FORTH-Systeme aus gutem Grund kein Definitionswort für Arrays zur Verfügung.
Weil aber der Aufbau dieser Definitionen nicht immer problemlos zu handhaben ist und die notwendigen Algorithmen in der Literatur manchmal falsch dargestellt werden, seien hier die Definitionen für eindimensionale Felder (Vektoren) und zweidimensionale Arrays angegeben.
Diese Datenstrukturen verbergen der zugrunde liegende Zellgröße in eigenen Definitionen, um die Worte ohne die typischen **2+ 2- 2*** schreiben zu können. Die

Basisgröße der Adressierung in einem 16bit-System ist das Byte, die Adressberechnungen arbeiten mit 16Bit-, also 2Byte-Adressen.

```
1 Constant byte
2 Constant integer
4 Constant double

| : bytes ;                  \ bytes ist 1*
| : integers  2* ;           \ Größe eines Bereiches
| : doubles   4 * ;

| : byte+     1+ ;           \ nächste byte-Adresse
| : integer+  2+ ;           \ nächste word-Adresse
| : double+   4+ ;
```

Obwohl FORTH als Sprache keine Typisierung der Daten erwartet, arbeitet die hier vorgestellte **FELD:**-Definition mit einer Typübernahme:

```
: Feld:
    Create ( #elements type - )
       dup c, *
       here over erase
       allot
    Does> ( index <addr>  - addr )
       count rot * + ;
```

Diese **FELD:**-Definition übergibt zur Laufzeit die Adresse des gegebenen Elementes, abhängig vom Typ der Variablen. Das Feld wird zur Compile-Zeit mit Nullen initialisiert.
In der möglichen Anwendung einer Meßdatenerfassung von Meßstellen wird die Handhabung von **FELD:** deutlich:

```
5    byte Feld: Bonn
5 integer Feld: Köln
5  double Feld: Moers

: .Werte  cr
     5 0 DO  I Köln   @   7 u.r  LOOP  cr;
```

Der Zugriff auf eine solche Datenstruktur kann z.B. über eine Schleife erfolgen, wobei - wie immer in FORTH - die zum Datentyp passenden Operatoren eingesetzt werden müssen.
Die Arbeit mit einer Matrix ist ähnlich, nur wird hier der Zugriff über zwei geschachtelte Schleifen erfolgen. Bitte beachten Sie, daß beim Zugriff FORTH-untypische Schleifenvariablen eingesetzt werden, die über dieSynonymdeklaration **Alias** definiert wurden.
Diese Definition von **MATRIX:** übernimmt ebenfalls eine Typangabe:

```
: Matrix:      \ #zeilen #spalten range (double|integer|byte)
               \ #zeile  #spalte  -- element_addr )
       Create
         2dup swap              \ integer|byte #zeile
         c, c,                  \ store #? and #?
         * * dup                \ amount bytes  amount bytes
           here swap erase      \ from here amount preset to 0
         allot                  \ amount allot
```

```
        Does>
          dup c@ 3 roll *
                 2 roll +
                 over byte+
              c@ * +
              integer+   ;

\ Drei Meßstationen haben je 4 Meßwerte erfaßt
\ und gespeichert

3 Constant #Stationen      ' I Alias Station#
4 Constant #Werte          ' J Alias Wert#

#Stationen #Werte byte Matrix: Messung

\ Wert|#Stat.|#Wert|Feldname|Operation
    3      0       0   Messung    c!
    6      0       1   Messung    c!
    9      0       2   Messung    c!
   12      0       3   Messung    c!

 5   1 0  Messung c!      7  2 0  Messung c!
10   1 1  Messung c!      9  2 1  Messung c!
15   1 2  Messung c!      9  2 2  Messung c!
10   1 3  Messung c!      7  2 3  Messung c!


: .all   ( -- )
#Stationen 0 DO cr
             #Werte 0 DO   Wert# Station#  Messung c@  3 .r
                      LOOP
             LOOP ;
```

Den einzelnen Operationen zum Einspeichern der Werte steht das Auslesen der Werte über zwei geschachtelte Schleifen gegenüber.

Eine weitere, oft genutzte Datenstruktur ist ein Ringpuffer (queue), *hier in der* Form eines 255 Byte langen counted strings. In diesen Puffer werden Zeichen hinten angehängt und vorne ausgelesen, so daß sich eine FIFO-Struktur ergibt:

```
Create QUEUE  0 c, 255 allot

: more?  ( addr -- n)  c@ ;

: q@     ( addr -- char)
     dup more? IF detract exit
               ELSE  ." Ringpuffer leer! " drop
               THEN ;

: q!     ( addr char --)   append ;

: q.     ( addr --)   count type ;

: qfill ( addr --)   $FF >expect ;
```

9.2 Methoden der objektorientierte Programmierung

Der **METHODS>**-Ansatz entbindet den Programmierer von der Pflicht, jedesmal den zum Datentyp passenden Operator auswählen zu müssen. Mit den generalisierten Operatoren **get put** und **show** wird die jeweils notwendige Zugriffsmethode aus einer Tabelle von Methoden ausgeführt .
Mittlerweile bildet sich allerdings als Bezeichnung für generalisierte Operatoren auch die Syntax **x@ x! xtype** ein.

```
\ Terry Rayburn   METHODS>                          euroFORML 87

: Methods>  [compile] Does>
        compile exit ; immediate restrict


\ early binding
  : [Method]:  Create 2* ,  immediate Does>
               here 2- @ @ 5 + swap @ + @ ,  ;

  0 [Method]: [get]   1 [Method]: [put]   2 [Method]: [show]


\ late binding
  : Method:  Create 2* ,
             Does> @ over 2- @ 5 + + perform ;

  0 Method: get   1 Method: put   2 Method: show
```

Der **METHODS>**-Ansatz wird in den Definitionen so eingesetzt:

```
: Integer:
    Variable   Methods>  @ ! u? ;

! : 2?  2@ d. ;
  : Double:
      2Variable  Methods>  2@ 2! 2? ;

: Queue: Create 0 c, 255 allot
         Methods>  q@ q! q.  ;

clear    \ löscht die Namen der Worte auf dem Heap
```

Später im Programm wird dann mit den generalisierten Methoden auf die Datenstrukturen zugegriffen:

```
Integer: Konto        1000 Konto put   Konto show
Double:  Moleküle     200.000 Moleküle put   Moleküle get d.
Queue:   Puffer       Puffer qfill
                      Ascii A  Puffer put
                      Puffer show
```

Die **Methods>**-Operatoren können mit den bekannten FORTH-Operatoren gemischt werden. Ebenso können die Tabelle beliebig geändert oder erweitert werden, so z.B. mit **4 Method: init** .
Soll diese Methode des Initialisierens nur bei **Integer:** eingesetzt werden, sind die anderen Tabellen entsprechend an der vierten Position aufzufüllen:

```
: Integer:
    Variable   Methods>  @ ! u? off ;

! : 2init   ( addr -- )  0. 2swap 2! ;
  : Double:
      2Variable  Methods>  2@ 2! 2? 2init ;

! : qinit   ( -- )
      ." Pufferinitialisierung noch nicht definiert! " ;
  : Queue: Create 0 c, 255 allot
           Methods>  q@ q! q. qinit ;
```

Der Zugriff erfolgt dann wie gezeigt mit Konto init oder Druckpuffer init . Die logische Konsequenz ist die Verbindung beider Möglichkeiten zu :METHODS> :

```
: :Methods>  [compile] :Does>  compile exit ; immediate
```

:Methods> erwartet eine Reihe von Operatoren und weist diese (Zugriffs-) Methoden dem zuletzt definierten Wort zu. Es wird analog zu :DOES> und METHODS> in dieser Form eingesetzt:

```
Create Puffer  0 c, 255 allot
               :Methods>  q@ q! q. qinit ;

Variable Zähler  :Methods>  @ ! u? off ;

Puffer show    Zähler init
```

1Ø. Manipulieren des Systemverhalten

Das grundlegende Systemverhalten ist bei den traditionellen Programmiersprachen in der Laufzeit-Bibliothek (runtime library) festgelegt.
Diese runtime library enthält die grundlegenden Routinen, die bei der Ausführung der Programme vorhanden sein müssen wie z.B. die Bildschirmansteuerung oder den Massenspeicherzugriff.

In FORTH entspricht das Programm KERNEL.COM dieser Laufzeit-Bibliothek. Auf diesen Systemkern wird dann die fertige Anwendung, also Ihr Programm, geladen. Ein Beispiel für eine solche Applikationserstellung ist das Erstellen des volks-FORTH-Arbeitssystems volks4th.com von der MS-DOS Kommandoebene aus:

```
A:>kernel include volks4th.sys
```

Im Gegensatz zu anderen Compilern, denen meist ein Assembler-Quelltext zugrunde liegt, ist KERNEL.COM aus einem FORTH-Quelltext durch METACOMPILATION erzeugt worden. Da sich ein FORTH-Quelltext besser pflegen und leichter ändern läßt als ein Assemblerprogramm, kann das grundlegende Systemverhalten in KERNEL.COM leicht neuen Erfordernissen anpassen.

1Ø.1 Patchen von FORTH-Befehlen

Möchte man ohne MetaCompilation das Verhalten von Funktion in KERNEL.COM ändern, so "patcht" man dieses Programm - eine Vorgehensweise, die WordStar-Nutzern gut bekannt ist. Soll beispielsweise die Funktion des Wortes . (DOT) im Kern geändert werden, ist so vorzugehen:

```
\ "patchen" von :-Definitionen im Kern, z.B. ".":

' . >body @  Constant altdot
: new.   ( n -- )  RDROP <new.action> ;

' new.   ' .   >body !   \ nun läuft die neue Version
altdot   ' .   >body !   \ und nun wieder die alte Version
```

Dabei ist es wichtig, die neue Arbeitsweise von . mit RDROP einzuleiten.

Im Gegensatz dazu können Sie die Eigenschaften Ihres volksFORTH-Arbeitssystems über den Inhalt von VOLKS4TH.SYS und durch direkte Änderung der Systemquell-texte individuell anpassen werden.

1Ø.2 Verwendung von DEFER-Wörtern

Eine weitere Möglichkeit, das Systemverhalten zur Laufzeit zu beeinflussen, wird in dem Wort **name** gezeigt:
Dort sorgt ein **exit** vor dem Definitions-abschließenden Semicolon für einen freien Platz im Wort. In diesen freien Platz kann später mit Hilfe des Wortes **'name** ein weiteres Wort "eingehängt" werden, das dann beim Aufruf von **name** mit ausgeführt wird. Auf diese Weise wird das Gesamtverhalten nicht vollständig geändert, sondern um eine weitere Funktion ergänzt.

Oft möchte man auch das Verhalten einer Anwendung an bestimmten Punkten umschaltbar machen. Dies wird in volksFORTH erreicht durch:

```
Defer <name>
```

So kann man schon auf ein Wort Bezug nehmen, bevor es definiert zu wurde. Dazu reicht es, volks4TH den Namen bekannt zu geben. Später können immer wieder andere Routinen an Stelle dieses Wortes ausgeführt werden:

```
' <action> Is <name>
( bitte auf das Häkchen  '  achten!)
```

Wird allerdings **<name>** ausgeführt, bevor es mit **IS** initialisiert wurde, so erscheint die Meldung "Crash" auf dem Bildschirm. Deshalb finden Sie oft diesen Ausdruck:

```
Defer <name>      ' noop Is <name>
```

So simpel ist die vektorielle Programmausführung in FORTH implementiert. Damit kann man zwei Probleme elegant lösen:

1. Das Wort <name> läßt sich bereits kompilieren, ohne daß ihm eine sinnvolle Aktion zugewiesen wurde. Damit ist die Kompilation von Wörtern möglich, die erst später definiert werden (Vorwärts-Referenzen).
2. Die nachträglich Veränderung von bereits kompilierten Wörtern ist damit möglich; vorausgesetzt, sie wurden als Defer-Wörter eingetragen. Wörter, die mit der Defer-Anweisung erzeugt wurden, lassen sich also in ihrem Ablaufverhalten nachträglich ändern. Diese Wörter behalten also ihren Namen bei, verändern aber ihr Verhalten.

Als Beispiel betrachten Sie bitte folgendes:

```
: Versuch#1   ." erster Versuch"  cr ;
: Versuch#2   ." zweiter Versuch"  cr ;

Defer Beispiel

' Versuch#1 Is Beispiel

Beispiel perform
```

Damit haben Sie jetzt **BEISPIEL** ausgeführt. Schauen Sie sich die Meldung an und geben ein:

```
' Versuch#2 Is Beispiel
```

Führen Sie erneut **Beispiel perform** aus - Sie bekommen nun die andere Meldung! Auf diese Art und Weise können Sie dem Wort **BEISPIEL** beliebige Funktionen zuweisen.

Die Kombination **DEFER/IS** behält also den Namen bei, ändert aber das Verhalten eines Wortes. Damit ist die Kombination **DEFER/IS** inhaltlich das Gegenteil von **ALIAS** , das den Namen ändert, aber die Funktion beibehält.

Mit **ALIAS** können Sie in volksFORTH einem Wort recht einfach einen neuen Namen geben. Sollte Ihnen das **V** des Editors nicht gefallen, benennen Sie es um in **WHERE** ! Dieses Umbenennen läßt sich auf zwei Arten durchführen, entweder im traditionellen FORTH-Stil mit

```
: where  v ;
```

oder dem volksFORTH gemäß mit

```
' v Alias where
```

10.3 Neudefinition von Befehlen

Sollen dagegen Worte des Compilers unter anderem Namen benutzt werden, so ist das in FORTH ebenfalls kein Thema:

```
: NeuName   AltName  ;
' THEN   Alias ENDIF      immediate restrict
: ENDIF  [compile] THEN  ;        immediate restrict
```

Eine Anwendung dieser Synonym-Deklarationen besteht darin, einen sogenannten Standard-Prolog zu verwirklichen: Damit Programme auch Wörter benutzen können, die der jeweilige Compiler nicht zur Verfügung stellt, schreibt man für den jeweiligen Compiler einen kleinen Vorspann (prelude) in FORTH83-Code zum Programm, der die jeweilig benötigten Definitonen enthält.
Oft stellt ein anderer Compiler eine identische Funktion unter einem anderen Namen zur Verfügung. So verwenden viele FORTH-Systeme **GOTOXY** statt des **AT** zur Cursorpositionierung. Nun ändert man im Prolog den Namen über die Alias-Funktion und erspart sich das quälende Suchen/Ersetzen im Editor.

10.4 DEFERed Wörter im volksFORTH83

Darüberhinaus besitzt das volksFORTH83 eine Reihe von Strukturen, die dazu dienen, das Verhalten des Systems zu ändern. Im System sind bereits folgende DEFERred Worte vorhanden:

R/W 'COLD 'RESTART 'ABORT 'QUIT NOTFOUND .STATUS DISKERR MAKEVIEW und **CUSTOM-REMOVE** .

Um sie zu ändern, benutzt man den Ausdruck:

```
' <action> Is <name>
```

Hierbei ist, wie oben besprochen, <name> ein durch **DEFER** erzeugtes Wort und <action> der Name des Wortes, das in Zukunft bei Aufruf von <name> ausgeführt wird.

Anwendungsmöglichkeiten dieser deferred Worte:

Durch Ändern von **R/W** kann man andere Floppies oder eine RAM-Disk betreiben. Es ist auch leicht möglich, Teile einer Disk gegen überschreiben zu schützen.

Durch Ändern von **NOTFOUND** kann man z.B. Worte, die nicht im Dictionary gefunden wurden, anschließend in einer Disk-Directory suchen lassen oder automatisch als Vorwärtsreferenz vermerken.

Ändert man **'COLD** , so kann man die Einschaltmeldung des volksFORTH83 unterdrücken und stattdessen ein Anwenderprogramm starten, ohne daß Eingaben von der Tastatur aus erforderlich sind (siehe z.B. "Erstellen einer Applikation").

Ähnliches gilt für **'RESTART** .

'ABORT ist z.B. dafür gedacht, eigene Stacks, z.B. für Fließkommazahlen, im Fehlerfall zu löschen. Die Verwendung dieses Wortes erfordert aber schon eine gewisse Systemkenntnis.

Das gilt auch für das Wort **'QUIT** . **'QUIT** wird dazu benutzt, eigene Quitloops in den Compiler einzubetten. Wer sich für diese Materie interessiert, sollte sich den Quelltext des Tracer anschauen. Dort wird vorgeführt, wie man das macht.

.STATUS schließlich wird vor Laden eines Blocks ausgeführt. Man kann sich damit anzeigen lassen, welchen Block einer längeren Sequenz das System gerade lädt und z.B. wieviel freier Speicher noch zur Verfügung steht.

CUSTOM-REMOVE kann vom fortgeschrittenen Programmierer dazu benutzt werden, eigene Datenstrukturen, die miteinander durch Zeiger verkettet sind, zu vergessen. Ein Beispiel dafür sind die File Control Blöcke des Fileinterfaces.

1Ø.5 Vektoren im volksFORTH83

Es gibt im System die Uservariable **ERRORHANDLER** . Dabei handelt es sich um eine normale Variable, die als Inhalt die Kompilationsadresse eines Wortes hat. Der Inhalt der Variablen wird auf folgende Weise ausgeführt:

```
errorhandler perform
```

Zuweisen und Auslesen dieser Variablen geschieht mit **@** und **!** . Der Inhalt von **ERRORHANDLER** wird ausgeführt, wenn das System **ABORT"** oder **ERROR"** ausführt und das von diesen Worten verbrauchte Flag wahr ist. Die Adresse des Textes mit der Fehlermeldung befindet sich auf dem Stack.
Siehe z.B. **(ERROR** .

Das volksFORTH83 benutzt die indirekten Vektoren **INPUT** und **OUTPUT** . Die Funktionsweise der sich daraus ergebenden Strukturen soll am Beispiel von **INPUT** verdeutlicht werden:

INPUT ist eine Uservariable, die auf einen Vektor zeigt, in dem wiederum vier Kompilationsadressen abgelegt sind. Jedes der vier Inputworte **KEY KEY? DECODE** und **EXPECT** führt eine der Kompilationsadressen aus. Kompiliert wird solch ein Vektor in der folgenden Form:

```
Input: vector   <name1> <name2> <name3> <name4> ;
```

Wird **VECTOR** ausgeführt, so schreibt er seine Parameterfeld-Adresse in die USERVariable **INPUT** . Von nun an führen die Inputworte die ihnen so zugewiesenen Worte aus. **KEY** führt also **<NAME1>** aus usw...

Das Beispiel **KEY?** soll dieses Prinzip verdeutlichen:

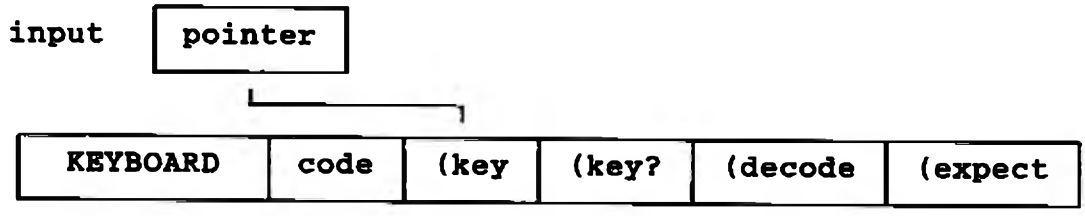
```
: key?   ( -- c )  input  @ 2+  perform ;
```

Tatsächlich wurde **KEY?** im System ebenso wie die anderen Inputworte durch das nicht mehr sichtbare definierende Wort **IN:** erzeugt. Ein Beispiel für einen Inputvektor, der die Eingabe von der Tastatur holt:

```
Input: keyboard
       (key (key? (decode (expect ;

keyboard
```

ergibt:



Analog verhält es sich mit **OUTPUT** und den Outputworten **EMIT CR TYPE DEL PAGE AT** und **AT?** . Outputvektoren werden mit **OUTPUT:** genauso wie die Inputvektoren erzeugt.
Bitte beachten Sie, daß immer alle Worte in der richtigen Reihenfolge aufgeführt werden müssen! Soll nur ein Wort geändert werden, so müssen sie trotzdem die anderen mit hinschreiben.

1Ø.6 Glossar

Defer (--)

ein definierendes Wort, das in der Form:

```
Defer <name>
```

benutzt wird. Erzeugt den Kopf für ein neues Wort <name> im Dictionary, hält 2 Byte in dessen Parameterfeld frei und speichert dort zunächst die Kompilationsadresse einer Fehlerroutine.
Wird <name> nun ausgeführt, so wird eine Fehlerbehandlung eingeleitet. Man kann dem Wort <name> jedoch zu jeder Zeit eine andere Funktion zuweisen mit der Sequenz:

```
' <action> Is <name>
```

Nach dieser Zuweisung verhält sich <name> wie <action>.
Mit diesem Mechanismus kann man zwei Probleme elegant lösen:
Einerseits läßt sich <name> bereits kompilieren, bevor ihm eine sinnvolle Aktion zugewiesen wurde. Damit ist die Kompilation erst später definierter Worte (Vorwärts-Referenzen) indirekt möglich.
Andererseits ist die Veränderung des Verhaltens von <name> für spezielle Zwecke auch nachträglich möglich, ohne neu kompilieren zu müssen.
Deferred Worte im System sind: **R/W 'COLD 'RESTART 'ABORT 'QUIT NOTFOUND .STATUS** und **DISKERR** . Diese Worte sind DEFERred, damit ihr Verhalten für die Anwendung geändert werden kann.

Is (cfa --)

ist ein Wort, mit dem das Verhalten eines deferred Wortes verändert werden kann. **IS** wird in der Form:

```
' <action> Is <name>
```

benutzt. Wenn <name> kein deferred Wort ist, wird eine Fehlerbehandlung eingeleitet, sonst verhält sich <name> anschließend wie <action>. Das Wort IS des volksFORTH83 kann innerhalb von Definitionen benutzt werden.
Siehe **DEFER** .

perform (addr --)

erwartet eine Adresse, unter der sich ein Zeiger auf die Kompilationsadresse eines Wortes befindet. Dieses Wort wird dann ausgeführt.
perform entspricht der Anweisungsfolge **@ execute** .

noop (--)

macht gar nichts.

Alias (cfa --)

ist ein definierendes Wort, das typisch in der Form:

' <oldname> Alias <newname>

benutzt wird. **ALIAS** erzeugt einen Kopf für <newname> im Dictionary. Wird <newname> aufgerufen, so verhält es sich wie <oldname>. Insbesondere wird beim Kompilieren nicht <newname>, sondern <oldname> im Dictionary eingetragen.
Im Unterschied zu **: <newname> <oldname> ;** ist es mit **ALIAS** möglich, Worte, die den Returnstack beeinflussen (z.B. **>r** oder **r>**), mit anderem Namen zu definieren. Außer dem neuen Kopf für <newname> wird kein zusaetzlicher Speicherplatz verbraucht. Gegenwärtig wird bei Ausführung von **>name** aus einer CFA in der Regel der letzte mit **ALIAS** erzeugte Name gefunden.

Input: (--) "input-colon"

ist ein definierendes Wort, benutzt in der Form:

Input: <name> newKEY newKEY? newDECODE newEXPECT ;

INPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen Satz von Zeigern auf Worte, die für die Eingabe von Zeichen zuständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld von <name> in die Uservariable **INPUT** geschrieben. Alle Eingaben werden jetzt über die neuen Eingabeworte abgewickelt.
Die Reihenfolge der Worte nach **INPUT: <name>** bis zum Semicolon muß eingehalten werden:

..key ..key? ..decode ..expect

Im System ist das mit **INPUT:** definierte Wort **keyboard** enthalten, nach dessen Ausführung alle Eingaben von der Tastatur geholt werden.
Siehe **DECODE** und **EXPECT** .

Output: (--) "output-colon"

ist ein definierendes Wort, benutzt in der Form:

```
Output: <name>
    newEMIT newCR newTYPE newDEL newPAGE  newAT newAT? ;
```

OUTPUT: erzeugt einen Kopf für <name> im Dictionary und kompiliert einen Satz von Zeigern auf Worte, die für die Ausgabe von Zeichen zuständig sind. Wird <name> ausgeführt, so wird ein Zeiger auf das Parameterfeld von <name> in die Uservariable OUTPUT geschrieben. Alle Ausgaben werden jetzt über die neuen Ausgabeworte abgewickelt.
Die Reihenfolge der Worte nach OUTPUT: <name> muß eingehalten werden:

```
..emit ..cr ..type ..del ..page ..at ..at?
```

Im System ist das mit OUTPUT: definierte Wort DISPLAY enthalten, nach dessen Ausführung alle Ausgaben auf den Bildschirm geleitet werden.
Vergleiche auch das Wort PRINT aus dem Printer-Interface.

'abort (--) "tick-abort"

ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in ABORT ausgeführt, bevor QUIT aufgerufen wird. Es kann benutzt werden, um "automatisch" selbstdefinierte Stacks zu löschen.

'cold (--) "tick-cold"

ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in COLD aufgerufen, bevor die Einschaltmeldung ausgegeben wird. Es wird benutzt, um Geräte zu initialisieren oder Anwenderprogramme automatisch zu starten.

'quit (--) "tick-quit"

ist ein deferred Wort, das normalerweise mit (QUIT besetzt ist. Es wird in QUIT aufgerufen, nachdem der Returnstack enleert und der interpretierende Zustand eingeschaltet wurde. Es wird benutzt, um spezielle Kommandointerpreter (wie z.B. im Tracer) aufzubauen.

'restart (--) "tick-restart"

Dies ist ein deferred Wort, das mit NOOP vorbesetzt ist. Es wird in RESTART aufgerufen, nachdem 'QUIT mit (QUIT besetzt wurde. Es wird benutzt, um Geräte nach einem Warmstart zu re-initialisieren.

(quit (--) "paren-quit"

Dieses Wort ist normalerweise der Inhait von 'QUIT . Es wird von QUIT benutzt. Es akzeptiert eine Zeile von der aktuellen Eingabeeinheit, führt sie aus und druckt "ok" bzw. "compiling".

.status (--) "dot-status"

Dieses Wort ist ein deferred Wort, das vor dem Einlesen einer Zeile bzw. dem Laden eines Blocks ausgeführt wird. Es ist mit NOOP vorbesetzt und kann dazu benutzt werden, Informationen über den Systemzustand oder den Quelltext auszugeben.

makeview (-- 16b)

ist ein deferred Wort, dem mit IS ein Wort zugewiesen wurde. Dieses Wort erzeugt aus dem gerade kompilierten Block eine 16b Zahl, die von Create in das Viewfeld eingetragen wird. Das deferred Wort wird benötigt, um das Fileinterface löschen zu können.

11. Vokabular-Struktur

Eine Liste von Worten ist ein Vokabular. Ein FORTH-System besteht im allgemeinen aus mehreren Vokabularen, die nebeneinander existieren. Neue Vokabulare werden durch das definierende Wort **VOCABULARY** erzeugt und haben ihrerseits einen Namen, der in einer Liste enthalten ist. Gewöhnlich kann von mehreren Worten mit gleichem Namen nur das zuletzt definierte erreicht werden. Befinden sich jedoch die einzelnen Worte in verschiedenen Vokabularen, so bleiben sie einzeln erreichbar.

11.1 Die Suchreihenfolge

Die Suchreihenfolge gibt an, in welcher Reihenfolge die verschiedenen Vokabulare nach einem Wort durchsucht werden.

Sie besteht aus zwei Teilen, dem auswechselbaren und dem festen Teil. Der auswechselbare Teil enthält genau ein Vokabular. Dies wird zuerst durchsucht. Wird ein Vokabular durch Eingeben seines Namens ausgeführt, so trägt es sich in den auswechselbaren Teil ein. Dabei wird der alte Inhalt überschrieben. Einige andere Worte ändern ebenfalls den auswechselbaren Teil. Soll ein Vokabular immer durchsucht werden, so muß es in den festen Teil übertragen werden. Dieser enthält null bis sechs Vokabulare und wird nur vom Benutzer bzw. seinen Worten verändert. Zur Manipulation stehen u.a. die Worte **ONLY ALSO TOSS** zur Verfügung. Das Vokabular, in das neue Worte einzutragen sind, wird durch das Wort **DEFINITIONS** angegeben. Die Suchreihenfolge kann man sich mit **ORDER** ansehen.

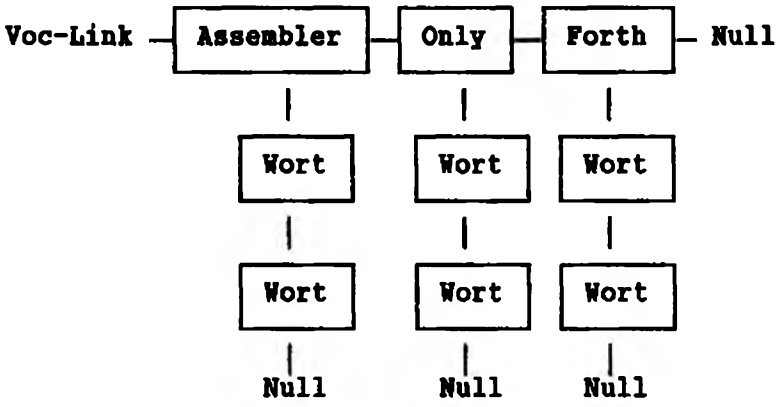
Ein Beispiele, das die Veränderung der Suchreihenfolge mit den eingegebenen Befehlen zeigt, wobei das Grundvokabular **ROOT** zwar in der Suchreihenfolge enthalten ist, aber nicht von **ORDER** angezeigt wird, um die Statuszeile nicht mit obligatorischen Informationen zu blockieren:

Eingabe:	ORDER ergibt dann:
Onlyforth	FORTH FORTH ROOT FORTH
Editor also	EDITOR EDITOR FORTH ROOT FORTH
Assembler	ASSEMBLER EDITOR FORTH ROOT FORTH
definitions Forth	FORTH EDITOR FORTH ROOT ASSEMBLER
: test ;	ASSEMBLER EDITOR FORTH ROOT ASSEMBLER

Hierbei ist vor allem auf colon (:) zu achten, das ebenfalls die Suchlaufpriorität ändert, indem es das Kompilationsvokabular **current** in den auswechselbaren Teil von **context** überträgt.

Der Inhalt eines Vokabulars besteht aus einer Liste von Worten, die durch ihre Linkfelder miteinander verbunden sind. Es gibt also genauso viele Listen wie Vokabulare. Alle Vokabulare sind selbst noch einmal über eine Liste verbunden,

deren Anfang in **VOC-LINK** steht. Diese Verkettung ist nötig, um ein komfortables **FORGET** zu ermöglichen. Man bekommt beispielsweise folgendes Bild:



11.2 Glossar

Vocabulary (--) 83

ein definierendes Wort, das in der Form:

Vocabulary ‹name›

benutzt wird.

VOCABULARY erzeugt einen Kopf für ‹name›, das den Anfang einer neuen Liste von Worten bildet. Wird ‹name› ausgeführt, so werden bei der Suche im Dictionary zuerst die Worte in der Liste von ‹name› berücksichtigt. Wird das **VOCABULARY** ‹name› durch die Sequenz:

‹name› **definitions**

zum Kompilations-Vokabular, so werden neue Wort-Definitionen in die Liste von ‹name› gehängt. Vergleiche auch **CONTEXT CURRENT ALSO TOSS ONLY FORTH ONLYFORTH** .

context (-- addr)

addr ist die Adresse des auswechselbaren Teils der Suchreihenfolge. Sie enthält einen Zeiger auf das erste zu durchsuchende Vokabular.

current (-- addr)

addr ist die Adresse eines Zeigers, der auf das Kompilationsvokabular zeigt, in das neue Worte eingefügt werden.

definitions (--) 83

ersetzt das gegenwärtige Kompilationsvokabular durch das Vokabular im auswechselbaren Teil der Suchreihenfolge, d.h. neue Worte werden in dieses Vokabular eingefügt.

Only (--)

Das Nennen dieses Vokabular löscht die Suchreihenfolge vollständig und ersetzt sie durch das Vokabular **ROOT** im festen und auswechselbaren Teil der Suchreihenfolge. **ROOT** enthält nur wenige Worte, die für die Erzeugung einer Suchreihenfolge benötigt werden.

also (--)

Ein Wort, um die Suchreihenfolge zu spezifizieren. Das Vokabular im auswechselbarem Teil der Suchreihenfolge wird zum ersten Vokabular im festen Teil gemacht, wobei die anderen Vokabulare des festen Teils nach hinten rücken. Ein Fehler liegt vor, falls der feste Teil sechs Vokabulare enthält.

toss (--)

entfernt das erste Vokabular des festen Teils der Suchreihenfolge. Insofern ist es das Gegenstück zu **ALSO** .

seal (--)

löscht das Vokabular **ROOT** , so daß es nicht mehr durchsucht wird. Dadurch ist es möglich, nur die Vokabulare des Anwenderprogramms durchsuchen zu lassen.

Onlyforth (--)

entspricht der häufig benötigten Sequenz:

```
ONLY FORTH ALSO DEFINITIONS
```

Forth (--) 83

Das ursprüngliche Vokabular.

Assembler (--)

Ein Vokabular, das Prozessor-spezifische Worte enthält, die für Code-Definitionen benötigt werden.

words (--)

gibt die Namen der Worte des Vokabulars, das im auswechselbaren Teil der Suchreihenfolge steht, aus, beginnend mit dem zuletzt erzeugtem Namen.

forth-83 (--) 83

Laut FORTH83-Standard soll dieses Wort sicherstellen, daß ein Standardsystem benutzt wird. Im volksFORTH funktionslos.

vp (-- addr) "v-p"

Eine Variable, die das Ende der Suchreihenfolge markiert. Sie enthält ausserdem Informationen über die Länge der Suchreihenfolge.

voc-link (-- addr)

Eine Uservariable, die den Anfang einer Liste mit allen Vokabularen enthält. Diese Liste wird u.a. für **FORGET** benötigt.

12. Dictionary-Struktur

Das FORTH-System besteht aus einem Dictionary von Worten. Die Struktur der Worte und des Dictionaries soll im folgenden erläutert werden.

12.1 Aufbau

Die FORTH-Worte sind in Listen angeordnet (s.a. Struktur der Vokabulare). Die vom Benutzer definierten Worte werden ebenfalls in diese Listen eingetragen.
Jedes Wort besteht aus sechs Teilen. Es sind dies:

1. block — Der Nummer des Blocks, in dem das Wort definiert wurde (siehe auch VIEW).
2. link — Die Adresse (Zeiger) namens lfa , die auf das "Link-feld" des nächsten Wortes zeigt.
3. count — Die Länge des Namens dieses Wortes und drei Markierungsbits. Die Adresse nfa zeigt auf dieses Byte, ebenso last .
4. name — Der Name selbst.
5. code — Eine Adresse (Zeiger), die auf den Maschinencode zeigt, der bei Aufruf dieses Wortes ausgeführt wird. Die Adresse dieses Feldes heißt Kompilationsadresse cfa .
6. parameter — Das Parameterfeld; die Adresse dieses Feldes heißt pfa .

Ein Wort sieht dann so aus :

Wort					
block	link	count	name	code	parameter ...

Im folgenden sollen diese sechs Felder einzeln detailliert betrachtet werden.

Block — Das Blockfeld enthält in codierter Form die Nummer des Blocks und den Namen des Files, in dem das Wort

definiert wurde. Wurde es von der Tastatur aus eingegeben, so enthält das Feld Null.

Link

Über das Linkfeld sind die Worte eines Vokabulars zu einer Liste verkettet. Jedes Link-Feld enthält die Adresse des vorherigen Link-Feldes. Jedes Wort zeigt also auf seinen Vorgänger. Das unterste Wort der Liste enthält im Link-Feld eine Null. Die Null zeigt das Ende der Liste an.

Countfeld

Das count field enthält die Länge des Namens (1..31 Zeichen) und drei Markierungsbits :

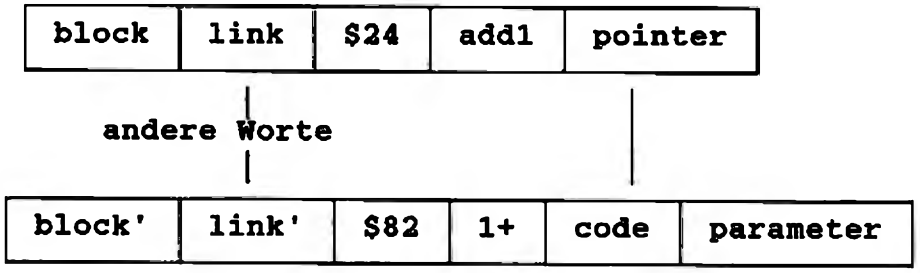
	restrict	immediate	indirect	Länge
Bit:	7	6	5	4..0

Ist das immediate-Bit gesetzt, so wird das entsprechende Wort im kompilierenden Zustand unmittelbar ausgeführt, und nicht ins Dictionary kompiliert (siehe auch **IMMEDIATE**) .

Ist das restrict-Bit gesetzt, so kann das Wort nicht durch Eingabe von der Tastatur ausgeführt, sondern nur in anderen Worten kompiliert werden. Gibt man es dennoch im interpretierenden Zustand ein, so erscheint die Fehlermeldung "compile only" (siehe auch **RESTRICT**).

Ist das indirect-Bit gesetzt, so folgt auf den Namen kein Codefeld, sondern ein Zeiger darauf. Damit kann der Name vom Rumpf (Code- und Parameterfeld) getrennt werden. Die Trennung geschieht z.B. bei Verwendung der Worte **!** oder **ALIAS** .

Beispiel: `' 1+ Alias add1`
ergibt folgende Struktur im Speicher (Dictionary) :



Name

Der Name besteht normalerweise aus ASCII-Zeichen. Bei der Eingabe werden Klein- in Großbuchstaben umgewandelt. Daher druckt **WORDS** auch nur großgeschriebene Namen.
Da Namen sowohl groß als auch klein geschrieben eingegeben werden können, haben wir eine Konvention erarbeitet, die die Schreibweise von Namen festlegt:
Bei Kontrollstrukturen wie **DO** **LOOP** etc. werden alle Buchstaben groß geschrieben.
Bei Namen von Vokabularen, immediate Worten und definierenden Worten, die **CREATE** ausführen, wird nur der erste Buchstabe groß geschrieben.
Beispiele sind: **Is FORTH Constant**
Alle anderen Worte werden klein geschrieben.
Beispiele sind: **dup cold base**
Bestimmte Worte, die von immediate Worten kompiliert werden, beginnen mit der öffnenden Klammer "(" , gefolgt vom Namen des immediate Wortes.
Ein Beispiel: **DO** kompiliert **(do** .
Diese Schreibweise ist nicht zwingend; Sie sollten sich aber daran halten, um die Lesbarkeit Ihrer Quelltexte zu erhöhen.

Code

Jedes Wort weist auf ein Stück Maschinencode. Die Adresse dieses Code-Stücks ist im Codefeld enthalten. Gleiche Worttypen weisen auf den gleichen Code. Es gibt verschiedene Worttypen, z.B. :-Definitionen, Variablen, Konstanten, Vokabulare usw. Sie haben jeweils ihren eigenen charakte- ristischen Code gemeinsam. Die Adresse des Code-Feldes heißt Kompilationsadresse.

Parameter

Das Parameterfeld enthält Daten, die vom Worttypen abhängen. Beispiele :

a) Typ "Constant"
Hier enthält das Parameterfeld des Wortes den Wert

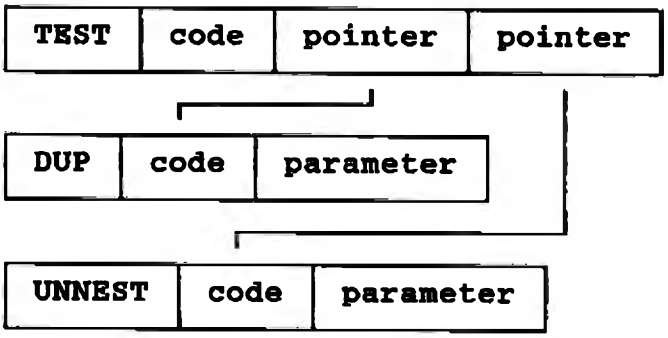
der Konstanten. Der dem Wort zugeordnete Code liest den Inhalt des Parameterfeldes aus und legt ihn auf den Stack.

b) Typ "Variable"
Das Parameterfeld enthält den Wert der Variablen, der zugeordnete Code liest jedoch nicht das Parameterfeld aus, sondern legt dessen Adresse auf den Stack. Der Benutzer kann dann mit dem Wort @ den Wert holen und mit dem Wort ! überschreiben.

c) Typ ":-definition"
Das ist ein mit : und ; gebildetes Wort. In diesem Fall enthält das Parameterfeld hintereinander die Kompilationsadressen der Worte, die diese Definition bilden. Der zugeordnete Code sorgt dann dafür, daß diese Worte der Reihe nach ausgeführt werden. Beispiel :

```
: test dup ;
```

ergibt:



Das Wort : hat den Namen **TEST** erzeugt. **UNNEST** wurde durch das Wort ; erzeugt

d) Typ "Code"
Worte vom Typ "Code" werden mit dem Assembler erzeugt. Hier zeigt das Codefeld in der Regel auf das Parameterfeld. Dorthin wurde der Maschinencode assembliert.
Codeworte im volksFORTH können leicht "umgepatcht" werden, da lediglich die Adresse im Codefeld auf eine

neue (andere) Maschinencodesequenz gesetzt werden muß.

12.2 Glossar

here (-- addr) 83
addr ist die Adresse des nächsten freien Dictionaryplatzes.

dp (-- addr) "d-p"
Eine Uservariable, die die Adresse des nächsten freien Dictionary-platzes enthält.

udp (-- addr) "u-d-p"
Eine Uservariable, in dem das Ende der bisher allokierten Userarea vermerkt ist.

allot (w --) 83
Allokiere w Bytes im Dictionary. Die Adresse des nächsten freien Dictionaryplatzes wird entsprechend verstellt.

uallot (n1 -- n2)
Allokiere bzw. deallokiere n1 Bytes in der Userarea. n2 gibt den Anfang des allokierten Bereiches relativ zum Beginn der Userarea an. Eine Fehlerbehandlung wird eingeleitet, wenn die Userarea voll ist.

c, (16b --) "c-comma"
ist ein **ALLOT** für ein Byte und speichert die unteren 8 Bit von 16b in **HERE 1-** .

, (16b --) 83 "comma"
ist **2 ALLOT** für 16b und speichere 16b ab **HERE 2-** .

' (-- addr) 83 "tick"
Wird in der Form ' <name> benutzt.
addr ist die Kompilationsadresse von <name>. Wird <name> nicht in der Suchreihenfolge gefunden, so wird eine Fehlerbehandlung eingeleitet.

name (--)

Eine Besonderheit von **name** ist, daß dieses Wort über zwei **EXIT**s verfügt. Zum einen das explizite **exit** und zum anderen das Semicolon, das ja auch ein **exit** ist. Die so einkompilierte Adresse kann dazu benutzt werden, beim Aufruf von **name** ein weiteres Wort ausführen zu lassen. Dieser Mechanismus kann als eine Variation von Deferred Worten aufgefaßt werden, weil auch hier das Wortverhalten geändert wird.
Siehe **'NAME** und auch **DEFER** .

'name (-- addr)

liefert die Bezugsadresse für das Einhängen eines Wortes, wie es bei **name** dargestellt wurde. Diese Möglichkeit, das Verhalten von Worten zu erweitern, wird bsp. im Editor eingesetzt. Dort wird im Wort **showload** ein Wort **show** über **'name** in **name** eingehängt und dann in **showoff** durch das Eintragen von **exit** wieder überschrieben, d.h. ausgehängt.

.name (addr --) "dot-name"

addr ist die Adresse des Countfeldes eines Namens. Dieser Name wird ausgedruckt. Befindet er sich im Heap, so wird das Zeichen ¦ vorangestellt. Ist addr Null, so wird "???" ausgegeben.

align (--)

macht nichts, weil der 8086/88 Prozessor auch von ungeraden Adressen Befehle holen kann, im Gegensatz zum 68000er. Diese Worte sind vorhanden, damit Sourcecode zwischen den Systemen transportabel ist.
Querverweis: **HALIGN** .

forget (--) 83

Wird in der Form **FORGET** <name> benutzt. Falls <name> in der Suchreihenfolge gefunden wird, so werden <name> und alle danach definierten Worte aus dem Dictionary entfernt. Wird <name> nicht gefunden, so wird eine Fehlerbehandlung eingeleitet. Liegt <name> in dem durch **SAVE** geschützten Bereich, so wird ebenfalls eine Fehlerbehandlung eingeleitet. Es wurden Vorkehrungen getroffen, die es ermöglichen, aktive Tasks und Vokabulare, die in der Suchreihenfolge auftreten, zu vergessen.

remove (dic sym thread -- dic sym)

Dies ist ein Wort, das zusammen mit **CUSTOM-REMOVE** verwendet wird. dic ist die untere Grenze des Dictionarybereiches, der vergessen werden soll und sym die obere. Typisch zeigt sym in den Heap. thread ist der Anfang einer Kette von Zeigern, die durch einen Zeiger mit dem Wert Null abgeschlossen wird. Wird **REMOVE** dann ausgeführt, so werden alle Zeiger (durch Umhängen der übrigen Zeiger) aus der Liste entfernt, die in dem zu vergessenden Dictionarybereich liegen. Dadurch ist es möglich, **FORGET** und ähnliche Worte auf Datenstrukturen anzuwenden.

(forget (addr --) "paren-forget"

Entfernt alle Worte, deren Kompilationsadresse oberhalb von addr liegt, aus dem Dictionary und setzt **HERE** auf addr. Ein Fehler liegt vor, falls addr im Heap liegt.

custom-remove (dic symb -- dic symb)

Ein deferred Wort, daß von **FORGET** , **CLEAR** usw. aufgerufen wird. dic ist die untere Grenze des Dictonaryteils, der vergessen wird und symb die obere. Gewöhnlich zeigt symb in den Heap.
Dieses Wort kann dazu benutzt werden, eigene Datenstrukturen, die Zeiger enthalten, bei **FORGET** korrekt abzuabrbeiten. Es wird vom Fileinterface verwendet, daher darf es nicht einfach überschrieben werden. Man kann es z.B. in folgender Form benutzen :

```
: <name> [ ' custom-remove >body @ , ]
    <liststart> @ remove ;
  ' <name> Is custom-remove
```

Auf diese Weise stellt man sicher, daß das Wort, das vorher in **CUSTOM-REMOVE** eingetragen war, weiterhin ausgeführt wird.
Siehe auch **REMOVE** .

save (--)

Kopiert den Wert aller Uservariablen in den Speicherbereich ab **ORIGIN** und sichert alle Vokabularlisten. Wird später **COLD** ausgeführt, so befindet sich das System im gleichen Speicherzustand wie bei Ausführung von **SAVE** .

empty (--)

Löscht alle Worte, die nach der letzten Ausführung von **SAVE** oder dem letzten Kaltstart definiert wurden. **DP** (und damit **HERE**)wird auf seinen Kaltstartwert gesetzt und der Heap gelöscht.

last (-- addr)

Variable, die auf das Countfeld des zuletzt definierten Wortes zeigt.
Siehe auch **RECURSIVE** und **MYSELF** .

hide (--)

Entfernt das zuletzt definierte Wort aus der Liste des Vokabulars, in das es eingetragen wurde. Dadurch kann es nicht gefunden werden. Es ist aber noch im Speicher vorhanden. (s.a. **REVEAL LAST**)

reveal (--)

Trägt das zuletzt definierte Wort in die Liste des Vokabulars ein, in dem es definiert wurde.

origin (-- addr)

addr ist die Adresse, ab der die Kaltstartwerte der Uservariablen abgespeichert sind.

name> (addr1 -- addr2) "name-from"

addr2 ist die Kompilationsadresse, die mit dem Countfeld in addr1 korrespondiert.

>body (addr1 -- addr2) "to-body"

addr2 ist die Parameterfeldadresse, die mit der Kompilationsadresse addr1 korrespondiert.

>name (addr1 -- addr2) "to-name"

addr2 ist die Adresse eines Countfeldes, das mit der Kompilationsadresse addr1 korrespondiert. Es ist möglich, daß es mehrere addr2 für ein addr1 gibt. In diesem Fall ist nicht definiert, welche ausgewählt wird.

In der Literatur finden Sie für die Umrechnung von Feld-Adressen auch oftmals folgende Worte:

```
: >link     >name 2-  ;
: link>     2+ name>  ;
: n>link    2-  ;
: l>name    2+  ;
```

13. Der HEAP

Eines der ungewöhnlichen und fortschrittlichen Konzepte des volksFORTH83 besteht in der Möglichkeit, Namen von Worten zu entfernen, ohne den Rumpf zu vernichten (headerless words).

Das ist insbesondere während der Kompilation nützlich, denn Namen von Worten, deren Benutzung von der Tastatur aus nicht sinnvoll wäre, tauchen am Ende der Kompilation auch nicht mehr im Dictionary auf. Man kann dem Quelltext sofort ansehen, ob ein Wort für den Gebrauch außerhalb des Programmes bestimmt ist oder nicht.

Die Namen, die entfernt wurden, verbrauchen natürlich keinen Speicherplatz mehr. Damit wird die Verwendung von mehr und längeren Namen und dadurch auch die Lesbarkeit gefördert.

Namen, die später eliminiert werden sollen, werden durch das Wort ¦ gekennzeichnet. Das Wort ¦ muß unmittelbar vor dem Wort stehen, das den zu eliminierenden Namen erzeugt. Der so erzeugte Name wird in einem besonderen Speicherbereich, dem Heap, abgelegt. Der Heap kann später mit dem Wort **CLEAR** gelöscht werden. Dann sind natürlich auch alle Namen, die sich im Heap befanden, verschwunden. Das folgende Beispiel soll eine Art Taschenrechner darstellen:

```
¦ Variable sum   1 sum !
```

Es werden weitere Worte definiert und dann **CLEAR** ausgeführt:

```
: clearsum   ( -- )   0 sum ! ;
: add        ( n -- ) sum +! ;
: show       ( -- )   sum @ . ;

clear
```

Diese Definitionen liefern die Worte **CLEARSUM** **ADD** und **SHOW** , während der Name der Variablen **SUM** durch **CLEAR** entfernt wurde. Das Codefeld und der Wert 0001 existieren jedoch noch. Man kann den Heap auch dazu "mißbrauchen", Code, der nur zeitweilig benötigt wird, nachher wieder zu entfernen. Der Assembler wird auf diese Art geladen, so daß er nach Fertigstellen der Applikation mit **CLEAR** wieder entfernt werden kann und keinen Platz im Speicher mehr benötigt.

Glossar

¦ (--) "headerless"
setzt bei Ausführung **?HEAD** so, daß der nächste erzeugte Name nicht im normalen Dictionaryspeicher angelegt wird, sondern auf dem Heap.

clear (--)

löscht alle Namen und Worte im Heap, so daß vorher mit ¦ definierte Worte nicht mehr benutzt werden können. Auf diese Weise kann der Geltungsbereich von Prozeduren eingeschränkt werden.

heap (-- addr)

Hier ist addr der Anfang des Heap. Er wird z.B. durch **HALLOT** geändert.

hallot (n --)

allokiert bzw. deallokiert n Bytes auf dem Heap. Dabei wird der Stack verschoben, ebenso wie der Beginn des Heap.

heap? (addr -- flag) "heap-question"

übergibt ein wahres Flag, wenn addr ein Byte im Heap adressiert, ansonsten **FALSE** .

?head (-- addr) "question-head"

ist eine Variable, die angibt, ob und wieviele der nächsten zu erzeugenden Namen im Heap angelegt werden sollen.
Siehe auch ¦ .

halign (--) "h-align"

dient nur der Softwarekompatibilität zu den anderen volks4TH-Systemen, da der 8Ø86-Prozessor nicht align-ed werden braucht.
Querverweis: **ALIGN** .

14. Die Ausführung von FORTH-Worten

Der geringe Platzbedarf übersetzter FORTH-Worte rührt wesentlich von der Existenz des Adressinterpreters her.
Wie im Kapitel über die Dictionary-Struktur beschrieben, besteht eine :-Definition aus dem Codefeld und dem Parameterfeld. Im Parameterfeld steht eine Folge von Adressen. Ein Wort wird kompiliert, indem seine Kompilationsadresse dem Parameterfeld der :-Definition angefügt wird. Eine Ausnahme bilden die Immediate-Worte. Da sie während der Kompilation ausgeführt werden, können sie dem Parameterfeld der :-Definition alles mögliche hinzufügen. Daraus wird klar, daß die meisten Worte innerhalb der :-Definition nur eine Adresse, also in einem 16-Bit-System genau 2 Bytes an Platz verbrauchen. Wird die :-Definition nun aufgerufen, so sollen alle Worte, deren Kompilationsadresse im Parameterfeld stehen, ausgeführt werden. Das besorgt der Adressinterpreter.

14.1 Der Aufbau des Adressinterpreters

Beim volksFORTH83 benutzt der Adressinterpreter einige Register der CPU, die im Kapitel über den Assembler aufgeführt werden. Es gibt aber mindestens die folgenden Register :

IP	ist der Instruktionszeiger (Instructionpointer). Er zeigt auf die nächste auszuführende Instruktion. Das ist beim volks-FORTH83 die Speicherzelle, die die Kompilationsadresse des nächsten auszuführenden Wortes enthält.
W	ist das Wortregister. Es zeigt auf die Kompilationsadresse des Wortes, das gerade ausgeführt wird.
SP	ist der (Daten-) Stackpointer. Er zeigt auf das oberste Element des Stacks.
RP	ist der Returnstackpointer. Er zeigt auf das oberste Element des Returnstacks.

14.2 Die Funktion des Adressinterpreters

NEXT ist die Anfangsadresse der Routine, die die Instruktion ausführt, auf die IP gerade zeigt. Die Routine **NEXT** ist ein Teil des Adressinterpreters. Zur Verdeutlichung der Arbeitsweise ist dieser Teil hier in High Level geschrieben:

```
Variable IP
Variable W

: Next
```

```
IP @  @  W !
2 IP +!
W perform ;
```

Tatsächlich ist **NEXT** jedoch eine Maschinencoderoutine, weil dadurch die Ausführungszeit von FORTH-Worten erheblich kürzer wird. **NEXT** ist somit ein Makro, die diejenige Instruktion ausführt, auf die das Register IP zeigt.
Ein Wort wird ausgeführt, indem der Code, auf den die Kompilations-adresse zeigt, als Maschinencode angesprungen wird. Der Code kann z.B. den alten Inhalt des IP auf den Returnstack bringen, die Adresse des Parameterfeldes im IP speichern und dann **NEXT** anspringen. Diese Routine gibt es wirklich, sie heißt "docol" und ihre Adresse steht im Codefeld jeder :-Definition.
Das Gegenstück zu dieser Routine ist ein FORTH-Wort mit dem Namen **EXIT** . Dabei handelt es sich um ein Wort, das das oberste Element des Returnstacks in den IP bringt und anschließend **NEXT** anspringt. Damit ist dann die Ausführung der Colon-Definition beendet.

Beipiel :

```
: 2*   dup + ;
: 2.   2* . ;
```

a.) Ein Aufruf von

5 2.

von der Tastatur aus führt zu folgenden Situationen :

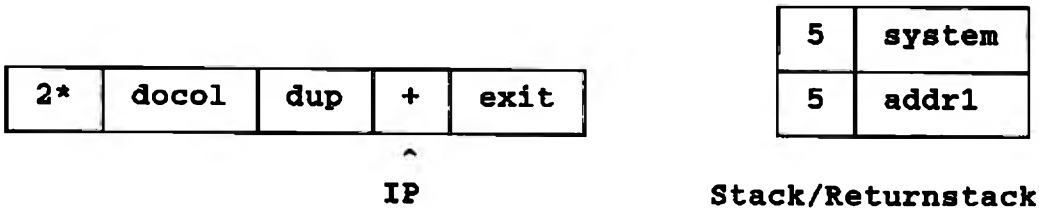
IP Stack/Returnstack

b.) Nach der ersten Ausführung von **NEXT** bekommt man :

addr1 Stack/Returnstack

IP Stack/Returnstack

c.) Nochmalige Ausführung von **NEXT** ergibt:
(**DUP** ist ein Wort vom Typ "Code")



d.) Nach der nächsten Ausführung von **NEXT** zeigt der IP auf **EXIT** und nach dem darauf folgenden **NEXT** wird addr1 wieder in den IP geladen:

e.) Die Ausführung von . erfolgt analog zu den Schritten b,c und d. Anschließend zeigt IP auf **EXIT** in 2. .
Nach Ausführung von **NEXT** kehrt das System wieder in den Text-interpreter zurück. Dessen Rückkehradresse wird durch system angedeutet. Damit ist die Ausführung von 2. beendet

14.3 Verschiedene IMMEDIATE-Worte

In diesem Abschnitt werden Beispiele für immediate Worte angegeben, die mehr als nur eine Adresse kompilieren.

a.) Zeichenketten (strings), die durch " abgeschlossen werden, liegen als counted strings im Dictionary vor.
Beispiel: `abort" Test"`
liefert: `| (ABORT" | 04 | T | e | s | t |`

b.) Einige Kontrollstrukturen kompilieren **?BRANCH** oder **BRANCH** , denen ein Offset mit 16 Bit Länge folgt.
Beispiel: `0< IF swap THEN`
liefert: `| 0< | ?BRANCH | 4 | SWAP |`
Beispiel: `BEGIN ?dup WHILE @ REPEAT`
liefert: `| ?DUP | ?BRANCH | 8 | @ | BRANCH | -10 |`

c.) Ebenso fügen **DO** und **?DO** einen Offset hinter das von ihnen kompilierte Wort **(DO** bzw. **(?DO** . Mit dem Dekompiler können Sie sich eigene Beispiele anschauen.

d.) Zahlen werden in das Wort **LIT** , gefolgt von einem 16-Bit-Wert, kompiliert.

Beispiel: [hex] 8 1000
liefert: ¦ CLIT ¦ $08 ¦ LIT ¦ $1000 ¦

14.4 Die Does>-Struktur

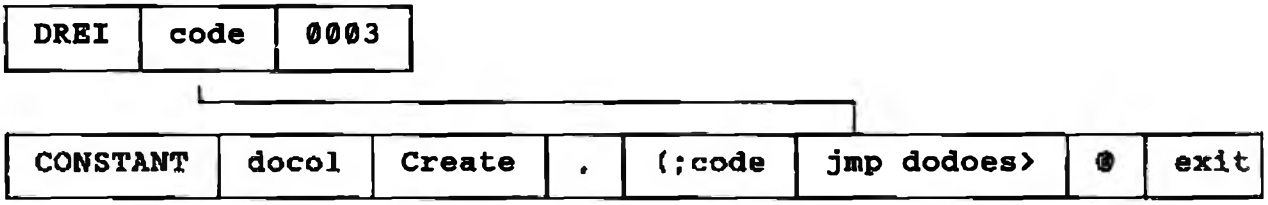
Die Struktur von Worten, die mit **CREATE .. DOES>** erzeugt wurden, soll anhand eines Beispiels erläutert werden.

Das Beispielprogramm lautet:

```
: Constant    ( <name> -- )
              ( -- n )
    Create , Does> @ ;

3 Constant drei
```

Der erzeugte Code sieht folgendermaßen aus:



Das Wort **(;CODE** wurde durch **DOES>** erzeugt. Es setzt das Codefeld des durch **CREATE** erzeugten Wortes **DREI** und beendet dann die Ausführung von **CONSTANT** . Das Codefeld von **DREI** zeigt anschließend auf **jmp dodoes>** . Wird **DREI** später aufgerufen , so wird der zugeordnete Code ausgeführt. Das ist in diesem Fall **jmp dodoes>** .

dodoes> legt nun die Adresse des Parameterfeldes von **DREI** auf den Stack und führt die auf **jmp dodoes>** folgende Sequenz von Worten aus. (Man beachte die Ähnlichkeit zu :-Definitionen). Diese Sequenz besteht aus **@** und **EXIT** .
Der Inhalt des Parameterfeldes von **DREI** wird damit auf den Stack gebracht. Der Aufruf von **DREI** liefert also tatsächlich 0003.

Statt des **jmp dodoes>** und der Sequenz von FORTH-Worten kann sich hinter **(;CODE** auch ausschließlich Maschinencode befinden. Das ist der Fall, wenn wir das Wort **;CODE** statt **DOES>** benutzt hätten. Wird diese Maschinencodesequenz später ausgeführt, so zeigt das W-Register des Adressinterpreters auf das Codefeld des Wortes, dem dieser Code zugeordnet ist. Erhöht man also das W-Register um 2 , so zeigt es auf das Parameterfeld des gerade auszuführenden Wortes.

14.5 Glossar

state (-- addr) 83

Eine Variable, die den gegenwärtigen Zustand enthält. Der Wert Null zeigt den interpretierenden Zustand an, ein von Null verschiedener Wert den kompilierenden Zustand.

>in (-- addr) 83 "to-in"

Eine Variable, die den Offset auf das gegenwärtige Zeichen im Quelltext enthält.
Siehe auch **WORD** .

source (-- addr +n)

liefert Anfang addr und maximale Länge +n des Quelltextes.
Ist BLK Null, beziehen sich Anfang und Länge auf den Text-eingabe-puffer, sonst auf den Block, dessen Nummer in **BLK** steht und der in den Rechnerspeicher kopiert wurde.
Siehe auch **BLOCK** und **>IN** .

blk (-- addr) 83 "b-l-k"

Eine Variable, die die Nummer des gerade als Quelltext interpretierten Blockes enthält. Ist der Wert von **BLK** Null, so wird der Quelltext vom Texteingabepuffer genommen.

load (n --) 83

Die Inhalte von **>IN** und **BLK** , die den gegenwärtigen Quelltext angeben, werden gespeichert. Der Block mit der Nummer n wird dann zum Quelltext gemacht. Der Block wird interpretiert. Die Interpretation wird bei Ende des Blocks abgebrochen, sofern das nicht explizit geschieht. Dann wird der alte Inhalt nach **BLK** und **>IN** zurückgebracht.
Siehe auch **BLK** , **>IN** und **BLOCK** .

((--) 83,I "paren"
(--) compiling

wird in der folgenden Art benutzt:

(ccc)

Die Zeichen ccc, abgeschlossen durch) , werden als Kommentar betrachtet. Kommentare werden ignoriert. Das Leerzeichen zwischen (und ccc ist nicht Teil des Kommentars. (kann im interpretierenden oder kompilierenden Zustand benutzt werden. Fehlt) , so werden alle Zeichen im Quelltext als Kommentar betrachtet.

\ (--) I "skip-line"
(--) compiling
Ignoriere den auf dieses Wort folgenden Text bis zum Ende der Zeile.
Siehe auch C/L .

\\ (--) I "skip-screen"
(--) compiling
Ignoriere den auf dieses Wort folgenden Text bis zum Ende des Blockes.
Siehe auch B/BLK .

\needs (--) "skip-needs"
wird in der folgenden Art benutzt:
\needs <name>
Wird <name> in der Suchreihenfolge gefunden, so wird der auf <name> folgende Text bis zum Ende der Zeile ignoriert. Wird <name> nicht gefunden, so wird die Interpretation hinter <name> fortgesetzt.

Beispiel: \needs Editor 1+ load
lädt den folgenden Block, falls EDITOR im Dictionary nicht vorhanden ist.

Ab der Version 3.81.3 wurde das Konzept um have eingeführt, so daß die bedingte Kompilation so eingesetzt wird:
have Editor not .IF 1+ load .THEN

In anderen Quellen finden Sie auch exists? .

+load (n --) "plus-load"
ladet den Block, dessen Nummer um n höher ist, als die Nummer des gegenwärtig interpretierten Blockes.

thru (n1 n2 --)
ladet die Blöcke von n1 bis inklusive n2.

+thru (n1 n2 --) "plus-thru"
ladet hintereinander die Blöcke, die n1..n2 vom gegenwärtigen Block entfernt sind.
Beispiel 1 2 +thru lädt die nächsten beiden Blöcke.

--> (--) I "next-block"
(--) compiling
Setze die Interpretation auf dem nächsten Block fort.

loadfile (-- addr)

addr ist die Adresse einer Variablen, die auf das FORTH-File zeigt, das gerade geladen wird. Diese Variable wird bei Aufruf von **LOAD** , **THRU** usw. auf das aktuelle File gesetzt.

bye (--)

Dieses Wort führt **FLUSH** und **EMPTY** aus. Anschließend wird der Monitor des Rechners angesprungen oder eine andere implementationsabhängige Funktion ausgeführt. Der Befehl dient zum Verlassen des volksFORTH.

cold (--)

Bewirkt den Kaltstart des Systems. Dabei werden alle nach der letzten Ausführung von **SAVE** definierten Worte entfernt, die Uservariablen auf den Wert gesetzt, den sie bei SAVE hatten, die Blockpuffer neu initialisiert, der Bildschirm gelöscht und die Einschaltmeldung

"volksFORTH-83 rev..."

ausgegeben. Anschliessend wird **RESTART** ausgeführt.

restart (--)

Bewirkt den Warmstart des Systems. Es setzt **'QUIT** , **ERRORHANDLER** und **'ABORT** auf ihre normalen Werte und führt **ABORT** aus.

interpret (--)

Beginnt die Interpretation des Quelltextes bei dem Zeichen, das durch den Inhalt von **>IN** indiziert wird. **>IN** indiziert relativ zum Anfang des Blockes, dessen Nummer in **BLK** steht. Ist **BLK** gleich Null, so werden Zeichen aus dem Texteingabepuffer interpretiert.

Auch **INTERPRET** benötigte bisher zur Implementation einen sehr mysteriösen Systempatch. Dank einer Idee von Mike Perry ist auch diese letzte Ecke nun abgeschliffen:
Das deferred Wort **PARSER** enthält entweder den Code für den Interpreter oder den Compiler (durch **[** und **]** umzuschalten) und Interpret ist nun eine **BEGIN...REPEAT** Schleife, in der das nächste Wort aus dem Quelltext geholt wird. Ist der Quelltext erschöpft, so wird die Schleife verlassen, andernfalls wird **PARSER** aufgerufen und dadurch das Wort entweder interpretiert oder compiliert.
Nun ist es auch sehr viel einfacher als vorher, selber eigene Worte zu definieren, die in **PARSER** eingehängt werden. Dies ist immer dann sinnvoll, wenn der Interpreter in einem Anwendungsprogramm anders als der übliche FORTH-Interpreter arbeiten soll.

prompt (--)

ist ein deferred Wort, das für die Ausgabe des "ok" verantwortlich ist. Es wurde auch das Wort (QUIT implementiert.
Nun ist es möglich, den FORTH-Interpreter auch wie ein "klassisches" Betriebssystem arbeiten zu lassen, in dem eine Meldung nicht nach jeder Aktion hinter der Eingabezeile ausgegeben wird, sondern vor einer Aktion am Anfang der Eingabezeile. Ein entsprechendes Beispiel befindet sich im Quelltext hinter der Definition von QUIT .

parser siehe INTERPRET.

word (char -- addr) 83

erzeugt einen counted String durch Lesen von Zeichen vom Quelltext, bis dieser erschöpft ist oder der Delimiter char auftritt.
Der Quelltext wird nicht zerstört. Führende Delimiter werden ignoriert. Der gesamte String wird im Speicher beginnend ab Adresse addr als eine Sequenz von Bytes abgelegt. Das erste Byte enthält die Länge des Strings (Ø..255). Der String wird durch ein Leerzeichen beendet, das nicht in der Längenangabe enthalten ist. Ist der String länger als 255 Zeichen, so ist die Länge undefiniert. War der Quelltext schon erschöpft, als WORD aufgerufen wurde, so wird ein String der Länge Null erzeugt. Wird der Delimiter nicht im Quelltext gefunden, so ist der Wert von >IN die Länge des Quelltextes. Wird der Delimiter gefunden, so wird >IN so verändert, dass >IN das Zeichen hinter dem Delimiter indiziert. #TIB wird nicht verändert. Der String kann sich oberhalb von HERE befinden.

parse (char -- addr +n)

liefert die Adresse addr und Länge +n des nächsten Strings im Quelltext, der durch den Delimiter char abgeschlossen wird.
+n ist Null, falls der Quelltext erschöpft oder das erste Zeichen char ist. >IN wird verändert.

name (-- addr)

holt den nächsten String, der durch Leerzeichen eingeschlossen wird, aus dem Quelltext, wandelt ihn in Grossbuchstaben um und hinterlässt die Adresse addr, ab der der String im Speicher steht.
Siehe auch WORD .

find (addr1 -- addr2 n) 83

addr1 ist die Adresse eines counted string. Der String enthält einen Namen, der in der aktuellen Suchreihenfolge gesucht wird.
Wird das Wort nicht gefunden, so ist addr2 = addr1 und n = Null. Wird das Wort gefunden, so ist addr2 dessen Kompilations-adresse und n erfüllt folgende Bedingungen:
n ist positiv, wenn das Wort immediate ist, sonst negativ.
n ist vom Betrag 2 , falls das Wort restrict ist, sonst vom Betrag 1 .

notfound (addr --)

Ein deferred Wort, das aufgerufen wird, wenn der Text aus dem Quelltext weder als Name in der Suchreihenfolge gefunden wurde, noch als Zahl interpretiert werden kann.
Kann benutzt werden, um eigene Zahl- oder Stringeingabeformate zu erkennen. Ist mit **NO.EXTENSIONS** vorbesetzt. Dieses Wort bricht die Interpretation des Quelltextes ab und druckt die Fehlermeldung "?" aus.

quit (--) 83

Entleert den Returnstack, schaltet den interpretierenden Zustand ein, akzeptiert Eingaben von der aktuellen Eingabeeinheit und beginnt die Interpretation des eingegebenen Textes.

' (-- addr) 83 "tick"

wird in der Form benutzt:

```
' <name>
```

addr ist die Kompilationsadresse von <name>. Wird <name> nicht in der Suchreihenfolge gefunden, so wird eine Fehlerbehandlung eingeleitet.

['] (-- addr) 83,I,C "bracket-tick"
(--) compiling

wird in der folgenden Art benutzt:

```
['] <name>
```

Kompiliert die Kompilationsadresse von <name> als eine Konstante. Wenn die :-definition später ausgeführt wird, so wird addr auf den Stack gebracht. Ein Fehler tritt auf, wenn <name> in der Suchreihenfolge nicht gefunden wird.

compile (--) 83,C

Typischerweise in der folgenden Art benutzt:

```
: <name> ... compile <namex> ... ;
```

Wird <name> ausgeführt, so wird die Kompilationsadresse von <namex> zum Dictionary hinzugefügt und nicht ausgeführt. Typisch ist <name> immediate und <namex> nicht immediate.

[compile] (--) 83,I,C "bracket-compile"
(--) compiling

Wird in der folgenden Art benutzt:

```
[compile] <name>
```

Erzwingt die Kompilation des folgenden Wortes <name>. Damit ist die Kompilation von immediate-Worten möglich.

immediate (--) 83

Markiert das zuletzt definierte Wort als "immediate", d.h. dieses Wort wird auch im kompilierenden Zustand ausgeführt.

restrict (--)

Markiert das zuletzt definierte Wort als "restrict", d.h. dieses Wort kann nicht vom Textinterpreter interpretiert, sondern ausschließlich in anderen Worten kompiliert werden.

[(--) 83,I "left-bracket"
(--) compiling

Schaltet den interpretierenden Zustand ein. Der Quelltext wird sukzessive ausgeführt. Typische Benutzung siehe **LITERAL** .

] (--) 83,I "right-bracket"
(--) compiling

Schaltet den kompilierenden Zustand ein. Der Text vom Quelltext wird sukzessive kompiliert. Typische Benutzung siehe **LITERAL** .

Literal (-- 16b) 83,I,C
(16b --) compiling

Typisch in der folgenden Art benutzt:

```
[ 16b ] Literal
```

Kompiliert ein systemabhängiges Wort, so daß bei Ausführung 16b auf den Stack gebracht wird.

," (--) "comma-quote"

Speichert einen counted String im Dictionary ab **HERE** . Dabei wird die Länge des Strings in dessen erstem Byte, das nicht zur Länge hinzugezählt wird, vermerkt.

Ascii (-- char) I
(--) compiling

Wird in der folgenden Art benutzt:

```
Ascii ccc
```

wobei ccc durch ein Leerzeichen beendet wird. char ist der Wert des ersten Zeichens von ccc im benutzten Zeichensatz (gewöhnlich ASCII). Falls sich das System im kompilierenden Zustand befindet, so wird char als Konstante compiliert. Wird die :-definition später ausgeführt, so liegt char auf dem Stack.

Does> (-- addr) 83,I,C "does"
(--) compiling

Definiert das Verhalten des Wortes, das durch ein definierendes Wort erzeugt wurde. Wird in der folgenden Art benutzt:

```
: <typ.name>
      <defining time action>
          <create> <compiletime action>
          Does> <runtime action> ;
```

und später:

```
<typ.name> <instance.name>
```

wobei <create> **CREATE** ist oder ein anderes Wort, das **CREATE** ausführt. Zeigt das Ende des Wort-erzeugenden Teils des definierenden Wortes an. Beginnt die Kompilation des Codes, der ausgeführt wird, wenn <instance.name> aufgerufen wird. In diesem Fall ist addr die Parameterfeldadresse von <name>. addr wird auf den Stack gebracht und die Sequenz zwischen **DOES>** und ; wird ausgeführt.

:Does>

wird benutzt in der Form:

```
Create <name> :Does> ... ;
```

Folgende Definition liegt im volks4TH **:DOES>** zugrunde:

```
| : (does> here >r [compile] Does> ;

: :Does>   last @ 0= Abort" without reference"
     (does>  current @ context !  hide 0 ] ;
```

:DOES> legt das Laufzeit-Verhalten des zuletzt definierten Wortes durch den bis zum ; nachfolgenden Code fest. Das betreffende Wort wurde mit **Create** oder einem **Create** benutzenden Wort definiert. Eine Anwendung finden Sie bei den Datentypen.

recursive (--) I,C
(--) compiling

Erlaubt die rekursive Kompilation des gerade definierten Wortes in diesem Wort selbst. Ferner kann Code für Fehlerkontrolle erzeugt werden.
Siehe auch **MYSELF** und **RECURSE** .

15. Der Assembler

Für das volksFORTH stehen zwei Assembler zur Verfügung:
- Ein Assembler entsprechend dem Laxen&Perry F83
- und der eigentliche volks4TH-Assembler.

15.1 Registerbelegung

Im Assembler sind FORTH-gemäße Namen für die Register gewählt worden. Dabei ist die Zuordnung zu den Intel-Namen folgendermassen:

A <=> AX	A- <=> AL
	A+ <=> AH

C <=> CX	C- <=> CL
	C+ <=> CH

Register A und C sind zur allgemeinen Benutzung frei.

D <=> DX	D- <=> DL
	D+ <=> DH

Das D Register enthält das oberste Element des (Daten)-Stacks.

R <=> BX	R- <=> RL
	R+ <=> RH

Das B Register hält den Return_stack_pointer

Die weiteren Prozessor-Register werden wie folgt genutzt:

```
U <=> BP      User_area_pointer
S <=> SP      Daten_stack_pointer
I <=> SI      Instruction_pointer
W <=> DI      Word_pointer, im allgemeinen zur Benutzung frei.
```

D: <=> DS E: <=> ES S: <=> SS C: <=> CS

Alle Segmentregister werden beim Booten auf den Wert des Codesegments C: gesetzt und muessen, wenn sie "verstellt" werden, wieder auf C: zurueckgesetzt werden.

Assemblerdefinitione werden in FORTH mit code eingeleitet und mit der Sequenz Next end-code beendet:

```
Code sp! ( addr -- )   D S mov  D pop     Next   end-code
```

15.2 Registeroperationen

Returnstack

```
Code rp@ ( -- addr )   D push   R D mov   Next   end-code
Code rp! ( addr -- )   D R mov   D pop    Next   end-code

Code rdrop     R inc   R inc   Next   end-code restrict
Code >r  ( 16b -- )  R dec   R dec   D R ) mov   D pop
     Next  end-code restrict
Code r>  ( -- 16b )  D push   R ) D mov   R inc   R inc
     Next  end-code restrict

Code r@ ( -- 16b )   D push   R ) D mov   Next   end-code

Code execute ( acf -- )   D W mov   D pop
                          W ) jmp   end-code

Code perform ( addr -- )  D W mov  D pop
                          W ) W mov   W ) jmp  end-code

\ : perform   ( addr -- )      @ execute ;
```

Segmentregister

```
Code ds@ ( -- addr )   D push   D: D mov   Next   end-code
```

15.3 Besonderheiten im volksFORTH83

Nachfolgend sind einige interesante Besonderheiten beschrieben:

NEXT ist ein Makro, sog. in-line code; wenn NEXT kompiliert wird, werden die in NEXT enthaltenen Mnemnonics in die Definition eingetragen:

```
Assembler also   definitions
: Next    lods   A W xchg   W ) jmp
          there tnext-link @ T , H tnext-link ! ;
```

Das Wort EI hinterläßt durch here die Adresse von NEXT auf dem Stack. Diese Adresse wird in das Codefeld von NOOP geschrieben, so daß ein explizites NEXT unnötig wird.

```
| Code ei    sti     here      Next   end-code

Code noop        here 2- !   end-code


Create recover   Assembler
       R dec   R dec   I R ) mov   I pop   Next
end-code
```

RECOVER soll nie ausgeführt werden, nur die Adresse dieser Routine ist für den TargetCompiler von Interesse:

```
:  ;c:   0 T recover # call ] end-code H ;
```

In ähnlicher Form findet sich dann die endgültige Form von ;c: für das Target-System wieder:

```
: ;c:   recover # call last off  end-code  0 ] ;
```

15.4 Glossar

assembler (--)

ist das Vokabular, das die Assembler-Worte enthält. Die Ausführung dieses Wortes nimmt **ASSEMBLER** als context in die Suchreihenfolge mit auf.

code <name> (--)

leitet eine Assembler-Definiton ein und schaltet die Suchreihenfolge auf das Vokabular **Assembler** .
Als Beispiel:

```
Code sp@ ( -- addr )   D push   S D mov   Next   end-code
```

end-code (--)

beendet eine Assembler-Definition.

Label <name> (--)
(-- addr)

legt eine benannte Marke an und liefert eine Adresse zurück:

```
Code exit
Label >exit   R ) I mov   R inc   R inc   Next   end-code

Code unnest   >exit  here 2- !   end-code
Code ?exit  ( flag -- )
   D D or   D pop   >exit 0= ?]      [[  Next   end-code
Code 0=exit ( flag -- )
   D D or   D pop   >exit 0= not ?]  ]]   end-code

\ : ?exit ( flag -- )   IF rdrop THEN ;
```

>label <name> (--)
(-- addr)

kann als temporäre Konstante betrachtet werden, die sich mit **clear** löschen läßt. Ein von **>label** zurückgelieferter Wert wird als Literal kompiliert.

Beispiel:

```
Assembler
nop   5555 # jmp              here 2- >label >cold
nop   5555 # jmp              here 2- >label >restart
```

;code (--)

beendet den high level Teil eines defining words und leitet dessen Assembler-Teil ein. Wird benutzt in der Form:

```
: <name>   <Create> ... ;code  ...  end-code
```

;c: (--)

leitet den high level Teil einer Assembler-Definition ein. Das Wort wird typisch im Handhaben einer Fehlersituation eingesetzt, da dann Geschwindigkeit keine Rolle mehr spielt. Als Beispiel:

```
| : stackfull ( -- )     depth $20 > Abort" tight stack"
  reveal last? IF dup heap? IF name> ELSE 4- THEN
                  (forget
               THEN
  true Abort" dictionary full" ;

  Code ?stack    u' dp U D) A mov   S A sub  CS
    ?[ $100 # A add  CS ?[ ;c: stackfull ; Assembler  ]? ]?
    u' s0 U D) A mov   A inc   A inc   S A sub
    CS not ?[  Next  ]?  ;c: true Abort" stack empty" ;

\ : ?stack     sp@ here - $100 u< IF  stackfull  THEN
\              sp@ s0 @ u> Abort" stack empty" ;
```

15.5 Kontrollstrukturen im Assembler

?[ ]?

benutzt in der Form, wobei cc ein condition code ist:

```
cc IF ... THEN
```

Als Beispiel:

```
Code dup    ( 16b -- 16b 16b )  D push   Next   end-code
Code ?dup   ( 16b -- 16b 16b / false )
   D D or   0= not ?[  D push  ]?  Next   end-code
```

?[ ][ ]?

benutzt in der Form:

```
cc IF ... ELSE ... THEN
```

[[ ]]

benutzt in der Form:

```
BEGIN ... REPEAT
```

Als Beispiel:

```
Code 1+ ( n1 -- n2 )    [[   D inc   Next
Code 2+ ( n1 -- n2 )    [[   D inc   swap ]]
Code 3+ ( n1 -- n2 )    [[   D inc   swap ]]
Code 4+ ( n1 -- n2 )    [[   D inc   swap ]]
! Code 6+ ( n1 -- n2 )   D inc   D inc   ]]  end-code

Code 1- ( n1 -- n2 )    [[   D dec   Next
Code 2- ( n1 -- n2 )    [[   D dec   swap ]]
```

[[cc]]?

benutzt in der Form:

```
BEGIN ... cc UNTIL
```

[[cc ?[[ ]]?

benutzt in der Form:

```
BEGIN ... cc WHILE ... REPEAT
```

15.6 Beispiele aus dem volksFORTH

pop

```
Code drop   ( 16b -- )   D pop   Next   end-code
```

```
  Code !  ( 16b addr -- )  D W mov   W ) pop   D pop
       Next  end-code

  Code +!      ( 16b addr -- )
     D W mov   A pop   A W ) add   D pop   Next   end-code


                                  push

  Code 2dup ( 32b -- 32b 32b )
     S W mov   D push   W ) push   Next   end-code

\ : 2dup ( 32b -- 32b 32b ) over over ;

  Code over   ( 16b1 16b2 -- 16b1 16b2 16b1 )
     A D xchg   D pop   D push   A push   Next   end-code


                                  mov

  Code @  ( addr -- 16b )  D W mov   W ) D mov
       Next   end-code

  Code c@   ( addr -- 8b )
     D W mov   W ) D- mov   0 # D+ mov   Next   end-code

  Code c!   ( 16b addr -- )
     D W mov   A pop   A- W ) mov   D pop   Next   end-code


                                  xchg

  Code flip  ( 16b1 -- 16b1' )   D- D+ xchg   Next   end-code

\    : flip  ( 16b1 -- 16b1' )  $100 um* or  ;


                                  neg

  Code negate  ( n1 -- n2 )    D neg   Next  end-code
\ : negate     ( n1 -- n2 )    not 1+ ;


                                  and

  Code and   ( 16b1 16b2 -- 16b3 )
     A pop   A D and   Next   end-code


                                  or

  Code or    ( 16b1 16b2 -- 16b3 )
     A pop   A D or   Next   end-code
\ : or       ( 16b1 16b2 -- 16b3 )   not swap not and not ;


                                  xor

  Code ctoggle  ( 8b addr -- )
     D W mov   A pop   A- W ) xor   D pop   Next   end-code
```

```
  Code xor   ( 16b1 16b2 -- 16b3 )
       A pop   A D xor   Next   end-code

\ : ctoggle  ( 8b addr -- )   under c@ xor swap c! ;


                                  add
  Code +   ( n1 n2 -- n3 )   A pop   A D add   Next   end-code


                                  sub
  Code -    ( n1 n2 -- n3 )
     A pop   D A sub   A D xchg   Next    end-code
\ : -    ( n1 n2 -- n3 )   negate + ;


                                  com
  Code not   ( 16b1 -- 16b2 )  D com   Next   end-code


                                  inc
  Code nip ( 16b1 16b2 -- 16b2 )  S inc  S inc   Next  end-code
\ : nip   swap drop ;

  Code 2drop ( 32b -- )  S inc   S inc   D pop   Next  end-code
\ : 2drop ( 32b -- ) drop drop ;


                                  sal
  Code pick    ( n -- 16b.n )
     D sal   D W mov   S W add   W ) D mov   Next   end-code
\ : pick    ( n -- 16b.n )     1+ 2* sp@ + @ ;


                        std  rep byte movs  cld
  Code roll  ( n -- )
     A I xchg   D sal   D C mov   D I mov   S I add
     I ) D mov   I W mov   I dec   W inc   std
     rep byte movs   cld   A I xchg   S inc   S inc   Next
  end-code
\ : roll   ( n -- )
\    dup >r  pick sp@ dup 2+  r> 1+ 2* cmove> drop ;


  Code -roll ( n -- )   A I xchg   D sal   D C mov
     S W mov   D pop   S I mov   S dec   S dec
     rep byte movs   D W ) mov   D pop   A I xchg   Next
  end-code

\ : -roll   ( n -- ) >r dup sp@ dup 2+
\    dup 2+ swap r@ 2* cmove r> 1+ 2* + ! ;
```

16. Der Multitasker

Ein wichtiger Aspekt der FORTH-Programmierung ist das Multitasking.

So wie man ein Programm in einzelne, unabhängige Module oder Worte aufteilt, so sollte man es auch in einzelne, unabhängige Prozesse aufteilen können. Das ist in den meisten Sprachen nicht möglich.

Das volksFORTH83 besitzt einen einfachen, aber leistungsfähigen Multitasker. Er ermöglicht die Konstruktion von Druckerspoolern, Uhren, Zählern und anderen einfachen Tasks, kann aber auch für Aufgaben eingesetzt werden, die über einen Druckerspooler hinausgehen.
Als Beispiel soll gezeigt werden, wie man einen einfachen Druckerspooler konstruiert. Das Programm für einen Druckerspooler lautet:

```
$F0 $100 Task background
: spool   background activate  1 100 pthru stop ;
multitask  spool
```

Normalerweise würde **PTHRU** den Rechner "lahmlegen", bis die Screens von 1 bis 100 ausgedruckt worden sind. Bei Aufruf von **SPOOL** ist das nicht so; der Rechner kann sofort weitere Eingaben verarbeiten. Damit alles richtig funktioniert, muß **PTHRU** allerdings einige Voraussetzungen erfüllen, die dieses Kapitel erklären will.

Das Wort **TASK** ist ein definierendes Wort, das eine Task erzeugt. Eine Task besitzt übrigens Userarea, Stack, Returnstack und Dictionary unabhängig von der sog. Konsolen- oder Main-Task.
Im Beispiel ist $F0 die Länge des reservierten Speicherbereichs für Returnstack und Userarea, $100 die Länge für Stack und Dictionary, jeweils in Bytes. Der Name der Task ist in diesem Fall **BACKGROUND** .

MULTITASK sagt dem Rechner, daß in Zukunft womöglich noch andere Tasks außer der Konsolentask auszuführen sind. Es schaltet also den Taskwechsler ein. Bei Ausführen von **SINGLETASK** wird der Taskwechsler abgeschaltet. Dann wird nur noch die gerade aktive Task ausgeführt.

Der neue Task tut nichts, bis sie aufgeweckt wird. Das geschieht durch das Wort **SPOOL** . Bei Ausführung von **SPOOL** geschieht nun folgendes:
Die Task **BACKGROUND** wird aufgeweckt und ihr wird der Code hinter **ACTIVATE** (nämlich **1 100 PTHRU STOP**) zur Ausführung übergeben. Damit ist die Ausführung von **SPOOL** beendet, es können jetzt andere Worte eingetippt werden. Die Task jedoch führt unverdrossen **1 100 PTHRU** aus, bis sie damit fertig ist. Dann stößt sie auf **STOP** und hält an. Man sagt, die Task schläft. Will man die Task während des Druckvorganges anhalten, z.B. um Papier nachzufüllen, so tippt man **BACKGROUND SLEEP** ein. Dann wird **BACKGROUND** vom Taskwechsler übergangen. Soll es weitergehen, so tippt man **BACKGROUND WAKE** ein.

Häufig möchte man erst bei Aufruf von **SPOOL** den Bereich als Argument angeben, der ausgedruckt werden soll. Das geht wie folgt:

```
: newspool ( from to -- ) 2 background pass pthru stop ;
```

Die Phrase **2 BACKGROUND PASS** funktioniert ähnlich wie **BACKGROUND ACTIVATE** , jedoch werden der Task auf dem Stack zusätzlich die beiden obersten Werte (hier from und to) übergeben. Um die Screens 1 bis 100 auszudrucken, tippt man jetzt ein:

```
1 100 newspool
```

Somit entspricht **BACKGROUND ACTIVATE** gerade der Phrase **0 BACKGROUND PASS** .

16.1 Implementation

Der Unterschied dieses Multitaskers zu herkömmlichen liegt in seiner kooperativen Struktur begründet. Damit ist gemeint, daß jede Task explizit die Kontrolle über den Rechner und die Ein/Ausgabegeräte aufgeben und damit für andere Tasks verfügbar machen muß. Jede Task kann aber selbst "wählen", wann das geschieht. Es ist klar, daß das oft genug geschehen muß, damit alle Tasks ihre Aufgaben wahrnehmen können.

Die Kontrolle über den Rechner wird durch das Wort **PAUSE** aufgegeben. **PAUSE** führt den Code aus, der den gegenwärtigen Zustand der gerade aktiven Task rettet und die Kontrolle des Rechners an den Taskwechsler übergibt. Der Zustand einer Task besteht aus den Werten des Interpreterpointers (IP) , des Returnstackpointers (RP) und des Stackpointers (SP).

Der Taskwechsler besteht aus einer geschlossenen Schleife. Jede Task enthält einen Maschinencodesprung auf die nächste Task, gefolgt von der Aufweckprozedur. Dort befindet sich die entsprechende Instruktion der nächsten Task. Ist die Task gestoppt, so wird dort ebenfalls ein Maschinencodesprung zur nächsten Task ausgeführt. Ist die Task dagegen aktiv, so ist der Sprung durch einen 1-Byte Call auf einen Vektor ersetzt worden, der die Aufweckprozedur auslöst. Diese Prozedur lädt den Zustand der Task (bestehend aus SP RP und IP) und setzt den Userpointer (UP), so daß er auf diese Task zeigt.

SINGLETASK ändert nun **PAUSE** so, daß überhaupt kein Taskwechsel stattfindet, wenn **PAUSE** aufgerufen wird. Das ist in dem Fall sinnvoll, wenn nur eine Task existiert, nämlich die Konsolentask, die beim Kaltstart des Systems "erzeugt" wurde. Dann würde **PAUSE** unnötig Zeit damit verbrauchen, einen Taskwechsel auszuführen, der sowieso wieder auf dieselbe Task führt.

Das System unterstützt den Multitasker, indem es während vieler Ein/Ausgabeoperationen wie **KEY** , **TYPE** und **BLOCK** usw. **PAUSE** ausführt. Häufig reicht das schon aus, damit eine Task (z.B. der Druckerspooler) gleichmäßig arbeitet.

Tasks werden im Dictionary der Konsolentask erzeugt. Jede besitzt ihre eigene Userarea mit einer Kopie der Uservariablen. Die Implementation des Systems wird aber durch die Einschränkung vereinfacht, daß nur die Konsolentask Eingabetext interpretieren bzw. kompilieren kann. Es gibt z.B. nur eine Suchreihenfolge, die im Prinzip für alle Tasks gilt. Da aber nur die Konsolentask von ihr Gebrauch macht, ist das nicht weiter störend.

Der Multitasker beim volksFORTH ist gegenüber z.B. dem des polyFORTH vereinfacht, da volksFORTH ist kein Multiuser-System ist. So besitzen alle Terminal-Einheiten (wir nennen sie Tasks) gemeinsam nur ein Lexikon und einen Eingabepuffer. Es darf daher nur der OPERATOR (wir nennen ihn Main- oder Konsolen-Task) kompilieren.

Es ist übrigens möglich, aktive Tasks mit **FORGET** usw. zu vergessen. Das ist eine Eigenschaft, die nicht viele Systeme aufweisen! Allerdings geht das manchmal auch schief... Nämlich dann, wenn die vergessene Task einen "Semaphor" (s.u.) besaß. Der wird beim Vergessen nämlich nicht freigegeben und damit ist das zugehörige Gerät blockiert.

Schließlich sollte man noch erwähnen, daß beim Ausführen eines Tasknamens der Beginn der Userarea dieser Task auf dem Stack hinterlassen wird.

16.2 Semaphore und Lock

Ein Problem, daß bisher noch nicht erwähnt wurde, ist die Frage, was passiert, wenn zwei Tasks gleichzeitig drucken (oder Daten von der Diskette lesen) wollen?

Es ist klar: Um ein Durcheinander oder Fehler zu vermeiden, darf das immer nur eine Task zur Zeit. Programmtechnisch wird das Problem durch "Semaphore" gelöst:

```
Variable disp  disp off
: newtype disp lock type disp unlock ;
```

Der Effekt ist der folgende: Wenn zwei Tasks gleichzeitig **NEWTYPE** ausführen, so kann doch nur eine zur Zeit **TYPE** ausführen, unabhängig davon, wie viele **PAUSE** in **TYPE** enthalten sind. Die Phrase **DISP LOCK** schaltet nämlich hinter der ersten Task, die sie ausführt, die "Ampel auf rot" und läßt keine andere Task durch. Die anderen machen solange **PAUSE** , bis die erste Task die Ampel mit **DISP UNLOCK** wieder auf grün umschaltet. Dann kann eine (!) andere Task die Ampel hinter sich umschalten usw. .

Übrigens wird die Task, die die Ampel auf rot schaltete, bei **DISP LOCK** nicht aufgehalten, sondern durchgelassen. Das ist notwendig, da ja **TYPE** ebenfalls **DISP LOCK** enthalten könnte (Im obigen Beispiel natürlich nicht, aber es ist denkbar).

Die Implementation sieht nun folgendermaßen aus, wobei man sich noch vor Augen halten muß, daß jede Task eindeutig durch den Anfang ihrer Userarea identifizierbar ist:

DISP ist ein sog. Semaphor; er muß den Anfangswert Ø haben!

LOCK schaut sich nun den Semaphor an: Ist er Null, so wird die gerade aktive Task (bzw. der Anfang ihrer Userarea) in den Semaphor eingetragen und die Task darf weitermarschieren.

Ist der Wert des Semaphors gerade die aktive Task, so darf sie natürlich auch weiter. Wenn aber der Wert des Semaphors von dem Anfang der Userarea der aktiven Task abweicht, dann ist gerade eine andere Task hinter der Ampel aktiv und die Task muß solange **PAUSE** machen, bis die Ampel wieder grün, d.h. der Semaphor null ist. **UNLOCK** muß nun nichts anderes mehr tun, als den Wert des Semaphors wieder auf Null setzen. **BLOCK** und **BUFFER** sind übrigens auf diese Weise für die Benutzung durch mehrere Tasks gesichert: Es kann immer nur eine Task das Laden von Blöcken von der Diskette veranlassen.

Eine Bemerkung bzgl. **BLOCK** und anderer Dinge:

Wie man dem Glossar entnehmen kann, ist immer nur die Adresse des zuletzt mit **BLOCK** oder **BUFFER** angeforderten Blockpuffers gültig, d.h. ältere Blöcke sind, je nach der Zahl der Blockpuffer, womöglich schon wieder auf die Diskette ausgelagert worden.

Auf der sicheren Seite liegt man, wenn man sich vorstellt, daß nur ein Blockpuffer im gesamten System existiert. Nun kann jede Task BLOCK ausführen und damit anderen Tasks die Blöcke "unter den Füßen" wegnehmen. Daher sollte man nicht die Adresse eines Blocks nach einem Wort, das PAUSE ausführt, weiter benutzen, sondern lieber neu mit BLOCK anfordern.

Das Beispiel

```
: .line ( block -- )
     block c/l bounds DO I c@ emit LOOP ;
```

ist falsch, denn nach **EMIT** stimmt der Adressbereich, den der Schleifenindex überstreicht, womöglich gar nicht mehr.

```
: .line ( block -- )
     c/l Ø DO  dup block I + c@ emit  LOOP drop ;
```

ist richtig, denn es wird nur die Nummer des Blocks, nicht die Adresse seines Puffers aufbewahrt.

```
: .line ( block -- ) block c/l type ;
```

ist falsch, da **TYPE** ja **EMIT** wiederholt ausführen kann und somit die von **BLOCK** gelieferte Adresse in **TYPE** ungültig wird.

```
: >type ( addr len -- )    pad place pad count type ;
```

ist multitasking-sicher, wenn ein String vom **BLOCK** geholt wird.

```
: .line ( block -- ) block c/l >type ;
```

ist deshalb richtig, denn PAD ist für jeden Task verschieden.

16.3 Glossar

activate (Taddr --) -tasker.scr-

aktiviert die Task, die durch Taddr gekennzeichnet ist, und weckt sie auf.
Vergleiche **SLEEP** , **STOP** , **PASS** , **PAUSE** , **UP@** , **UP!** und **WAKE** .

lock (semaddr --) -kernel.scr-

Der Semaphor (eine **VARIABLE**), dessen Adresse auf dem Stack liegt, wird von der Task, die **LOCK** ausführt, blockiert.
Dazu prüft **LOCK** den Inhalt des Semaphors. Zeigt der Inhalt an, daß eine andere Task den Semaphor blockiert hat, so wird **PAUSE** ausgeführt, bis der Semaphor freigegeben ist. Ist der Semaphor freigegeben, so schreibt **LOCK** das Kennzeichen der Task, die **LOCK** ausführt, in den Semaphor und sperrt ihn damit für alle anderen Tasks. Den FORTH-Code zwischen **semaddr LOCK** ... und ... **semaddr UNLOCK** kann also immer nur eine Task zur Zeit ausführen.
Semaphore schützen gemeinsame Ressourcen (z.B. den Drucker) vor dem gleichzeitigen Zugriff durch verschiedene Tasks.
Vergleiche **UNLOCK** und **RENDEZVOUS** .

multitask (--) -tasker.scr-

schaltet das Multitasking ein.
Das Wort **PAUSE** ist dann keine NOOP-Funktion mehr, sondern gibt die Kontrolle über die FORTH-Maschine an eine andere Task weiter.

pass (nØ .. nr-1 Taddr r --) -tasker.scr-

aktiviert die Task, die durch Taddr gekennzeichnet ist, und weckt sie auf. r gibt die Anzahl der Parameter nØ bis nr-1 an, die vom Stack der **PASS** ausführenden Task auf den Stack der durch Taddr gekennzeichneten Task übergeben werden. Die Parameter nØ bis nr-1 stehen dieser Task dann in der gleichen Reihenfolge auf ihrem Stack zur weiteren Verarbeitung zur Verfügung.
Vergleiche **ACTIVATE** .

pause (--) -kernel.scr-

ist eine NOOP-Funktion, wenn der Singletask-Betrieb eingeschaltet ist; bewirkt jedoch, nach Ausführung von MULTITASK , daß die Task, die PAUSE ausführt, die Kontrolle über die FORTH-Maschine an eine andere Task abgibt.
Existiert nur eine Task, oder schlafen alle anderen Tasks, so wird die Kontrolle unverzüglich an die Task zurückgegeben, die PAUSE ausführte. Ist mindestens eine andere Task aktiv, so wird die Kontrolle von dieser übernommen und erst bei Ausführung von PAUSE oder STOP in dieser Task an eine andere Task weitergegeben. Da die Tasks ringförmig miteinander verkettet sind, erhält die Task, die zuerst PAUSE ausführte, irgendwann die Kontrolle zurück. Eine Fehlerbedingung liegt vor, wenn eine Task weder PAUSE noch STOP ausführt.
Vergleiche STOP , MULTITASK und SINGLETASK .

rendezvous (semaddr --) -tasker.scr-

gibt den Semaphor (die VARIABLE) mit der Adresse semaddr frei (siehe UNLOCK) und führt PAUSE aus, um anderen Tasks den Zugriff auf das, durch diesen Semaphor geschützte, Gerät zu ermöglichen. Anschließend wird LOCK ausgeführt, um das Gerät zurück zu erhalten. Dies ist eine Methode, eine zweite Task nur an einer genau benannten Stelle laufen zu lassen.

singletask (--) -tasker.scr-

schaltet das Multitasking aus.
PAUSE ist nach Ausführung von SINGLETASK eine NOOP-Funktion.
Eine Fehlerbedingung besteht, wenn eine Hintergrund-Task SINGLETASK ohne anschließendes MULTITASK ausführt, da die Main- oder Terminal-Task dann nie mehr die Kontrolle bekommt.
Vergleiche UP@ und UP! .

sleep (Taddr --) -tasker.scr-

bringt die Task, die durch Taddr gekennzeichnet ist, zum Schlafen.
SLEEP hat den gleichen Effekt, wie die Ausführung von STOP durch die Task selbst. Der Unterschied ist, daß STOP in der Regel am Ende des Jobs der Task ausgeführt wird. SLEEP trifft die Task zu einem nicht vorhersehbaren Zeitpunkt, so daß die laufende Arbeit der Task abgebrochen wird.
Vergleiche WAKE .

stop (--) -tasker.scr-

bewirkt, daß die Task, die **STOP** ausführt, sich schlafen legt.
Wichtige Zeiger der FORTH-Maschine, die den Zustand der Task kennzeichnen, werden gerettet, dann wird die Kontrolle an die nächste Task abgegeben, deren Zeiger wieder der FORTH-Maschine übergeben werden, so daß diese Task ihre Arbeit an der alten Stelle aufnehmen kann. **PAUSE** führt diese Aktionen ebenfalls aus, der Unterschied zu **STOP** ist, daß die ausführende Task bei **PAUSE** aktiv, bei **STOP** hingegen schlafend hinterlassen wird.
Vergleiche **PAUSE** , **WAKE** und **SLEEP** .

Task (rlen slen <name> --) -tasker.scr-

ist ein definierendes Wort, das in der Form:

rlen slen Task <name>

benutzt wird. **TASK** erzeugt einen Arbeitsbereich für einen weiteren Job, der gleichzeitig zu schon laufenden Jobs ausgeführt werden soll.
Die Task erhält den Namen <name>, hat einen Stack-Bereich der Länge slen und einen Returnstack-Bereich der Länge rlen.
Im Stack Bereich liegen das Task-eigene Dictionary einschließlich **PAD** , das in Richtung zu höheren Adressen wächst, und der Daten-Stack, der zu niedrigen Adressen wächst. Im Returnstack-Bereich befinden sich die Task-eigene USER-Area (wächst zu höheren Adressen) und der Returnstack, der gegen kleinere Adressen wächst.
Eine Task ist ein verkleinertes Abbild des FORTH-Systems, allerdings ohne den Blockpuffer-Bereich, der von allen Tasks gemeinsam benutzt wird.
Zur Zeit ist es nicht zugelassen, daß Jobs einer Hintergrundtask kompilieren. Die Task ist nur der Arbeitsbereich für einen Hintergrund-Job, nicht jedoch der Job selbst.
Die Ausführung von <name> in einer beliebigen Task hinterläßt die gleiche Adresse, die die **Task <name>** selbst mit **UP@** erzeugt und ist zugleich die typische Adresse, die von **LOCK** , **UNLOCK** und **RENDEZVOUS** im Zusammenhang mit Semaphoren verwendet wird, bzw. von **ACTIVATE** , **PASS** , **SLEEP** und **WAKE** erwartet wird.

unlock (semaddr --) -kernel.scr-

gibt den Semaphor (die VARIABLE), dessen Adresse auf dem Stack ist, für alle Tasks frei. Ist der Semaphor im Besitz einer anderen Task, so wartet UNLOCK mit PAUSE auf die Freigabe. Vergleiche LOCK und die Beschreibung des Taskers.

up@ (-- Taddr) -kernel.scr- "u-p-fetch"
liefert die Adresse Taddr des ersten Bytes der USER-Area der Task, die **UP@** ausführt (siehe **TASK**). In der USER-Area sind Variablen und andere Datenstrukturen hinterlegt, die jede Task für sich haben muß.
Vergleiche **UP!** .

up! (addr --) -kernel.scr- "u-p-store"
richtet den **UP** (User Pointer) der FORTH-Maschine auf addr.
Vorsicht ist bei der Verwendung von **UP!** in Hintergrund-Tasks geboten.
Vergleiche **UP@** .

wake (Taddr --) -kernel.scr-
weckt die Task, die durch Taddr gekennzeichnet ist, auf.
Die Task führt ihren Job dort weiter aus, wo sie durch **SLEEP** angehalten wurde oder wo sie sich selbst durch **STOP** beendet hat (Vorsicht!).
Vergleiche **SLEEP** , **STOP** , **ACTIVATE** und **PASS** .

User (--) 83
ist ein definierendes Wort, benutzt in der Form:

```
User <name>
```

USER erzeugt einen Kopf für <name> im Dictionary und hält 2 Byte in der Userarea frei (siehe **UALLOT**).
Diese 2 Byte werden für den Inhalt der Uservariablen benutzt und werden nicht initialisiert. Im Parameterfeld der Uservariablen im Dictionary wird nur ein Offset zum Beginn der Userarea abgelegt. Wird <name> ausgeführt, so wird die Adresse des Wertes der Uservariablen in der Userarea auf den Stack gegeben. Uservariablen werden statt normaler Variablen z.B. dann benutzt, wenn der Einsatz des Multitaskers geplant ist und mindestens eine Task die Variable unbeeinflußt von anderen Tasks benötigt (Jede Task hat ihre eigene Userarea).

forget (--) 83
Wird in folgender Form benutzt:

```
FORGET <name>
```

Falls <name> in der Suchreihenfolge gefunden wird, so werden <name> und alle danach definierten Worte aus dem Dictionary entfernt. Wird <name> nicht gefunden, so wird eine Fehlerbehandlung eingeleitet. Liegt <name> in dem durch **SAVE** geschützten Bereich, so wird ebenfalls eine Fehlerbehandlung eingeleitet. Es wurden Vorkehrungen getroffen, die es ermöglichen, aktive Tasks und Vokabulare, die in der Suchreihenfolge auftreten, zu vergessen.

block (u -- addr) 83

addr ist die Adresse des ersten Bytes des Blocks u in dessen Blockpuffer. Der Block u stammt aus dem File in **ISFILE** .

BLOCK prüft den Pufferbereich auf die Existenz des Blocks Nummer u. Befindet sich der Block u in keinem der Blockpuffer, so wird er vom Massenspeicher in einen an ihn vergebenen Blockpuffer geladen. Falls der Block in diesem Puffer als **UPDATE**d markiert ist, wird er auf den Massenspeicher gesichert, bevor der Blockpuffer an den Block u vergeben wird. Nur die Daten im letzten Puffer, der über **BLOCK** oder **BUFFER** angesprochen wurde, sind sicher zugreifbar. Alle anderen Blockpuffer dürfen nicht mehr als gültig angenommen werden (möglicherweise existiert nur 1 Blockpuffer).

Vorsicht ist bei Benutzung des Multitaskers geboten, da eine andere Task **BLOCK** oder **BUFFER** ausführen kann. Der Inhalt eines Blockpuffers wird nur auf den Massenspeicher gesichert, wenn der Block mit **UPDATE** als verändert gekennzeichnet wurde.

17. Debugging-Techniken

Fehlersuche ist in allen Programmiersprachen die aufwendigste Aufgabe des Programmierers.

Für verschiedene Programme sind in der Regel auch verschiedene Hilfsmittel erforderlich. Daher kann dieses Kapitel die Fehler-suche nicht erschöpfend behandeln. Da aber Anfänger häufig typische Fehler machen, kann man gerade für diese Gruppe brauchbare Hilfsmittel angeben.

Voraussetzung für die Fehlersuche ist immer ein übersichtliches und verständliches Programm. In FORTH bedeutet das:

- suggestive und prägnante Namen für Worte
- starke Faktorisierung, d.h. sinnvoll zusammengehörende Teile eines Wortes sind zu einem eigenen Wort zusammengefaßt. Worte sollten durchschnittlich nicht mehr als 2 - 3 Zeilen lang sein !
- Übergabe von Parametern auf dem Stack statt in Variablen, wo immer es möglich ist.

Guter Stil in FORTH ist nicht schwer, erleichtert aber sehr die Fehlersuche. Ein Buch, das auch diesen Aspekt behandelt, sei unbedingt empfohlen : "In FORTH denken" von Leo Brodie, Hanser Verlag 1986.

Sind die genannten Bedingungen erfüllt, ist es meist möglich, die Worte interaktiv zu testen. Damit ist gemeint, daß man bestimmte Parameter auf den Stack legt und anschließend das zu testende Wort aufruft. Anhand der hinterlassenen Werte auf dem Stack kann man dann beurteilen, ob das Wort korrekt arbeitet.
Sinnvollerweise testet man natürlich die zuerst definierten Worte auch zuerst, denn ein Wort, das fehlerhafte Worte aufruft, funktioniert natürlich nicht korrekt. Wenn nun ein Wort auf diese Weise als fehlerhaft identifiziert wurde, muß man das Geschehen innerhalb des Wortes verfolgen. Das geschieht meist durch "tracen".

17.1 Der Tracer

Allgemein wird man zuerst das fehlerhafte Wort isoliert testen wollen. Dazu legt man ausgewählte Werte als Parameter für dieses Wort auf den Stack und beobachtet die Verarbeitung dieser Werte innerhablb des Wortes.

Angenommen, Sie wollen das Wort -TRAILING auf Fehler untersuchen. Die Funktion dieses Wortes ist das "Abschneiden" von Leerzeichen am Ende einer Zeichenkette:

```
: -trailing  ( addr1 n1 -- addr1 n2 )
2dup bounds ?DO 2dup + 1- c@ bl =
IF leave THEN 1- LOOP ;
```

Dabei könnten Sie dem Wort schon im Editor ein anderes Erscheinungsbild geben, indem Sie z.B. die Schleifenanweisungen **?DO** und **LOOP** bündig untereinander schreiben, durch das Einfügen von Leerzeichen eine Gliederung schaffen oder die **IF...THEN**-Anweisung auf einer eigenen Zeile isolieren.

Zum Testen des Wortes wird ein String benötigt. Sie definieren bitte:

```
Create teststring  ," Dies ist ein Test          "
```

Dabei haben Sie absichtlich einige zusätzliche Leerzeichen eingefügt.

Um die Werte auf dem Stack, meist Adressen, besser interpretieren zu können, schalten Sie mit **HEX** die Zahlenbasis um und geben nun ein (Antwort des Computers unterstrichen) :

```
hex ok
teststring .s 8686 ok
count      .s 1B 8687 ok
-trailing  .s 1B 8687 ok
```

Der Aufruf von **TESTSTRING** liefert Ihnen die Stringadresse (hexadezimal) 8686 und das Wort **COUNT** berechnet Ihnen daraus die Länge der Zeichenkette und ihren Beginn.: 1B 8687 (auch hexadezimal)
Beim Aufruf von **-TRAILING** stellen Sie an Hand der unveränderten Adressen zu Ihrem Erstaunen fest, daß **-TRAILING** kein einziges Leerzeichen abgeschnitten hat.

Spätestens jetzt sollten Sie am Rechner sitzen und den Tracer laden, wenn er noch nicht im System vorhanden ist. Prüfen Sie dazu, ob es das Wort **TOOLS** im FORTH-Vokabular gibt, dann ist der Tracer vorhanden. Denn der Tracer gehört zum Vokabular **Tools** , dessen Quelltexte Sie auf Ihrer Diskette finden.

Mit dem Tracer können Sie Worte, die mit dem **:** definiert wurden, schrittweise testen. Der Tracer läßt sich mit dem Wort **TRACE** starten, das seinerseits ein zu tracendes FORTH-Wort erwartet. **TRACE** schaltet im Gegensatz zu **DEBUG** nach Durchlauf des Wortes den Tracer automatisch mit **END-TRACE** wieder ab.

Um den Tracer zu benutzen, geben Sie nun folgendes ein:

```
teststring count
.s 1B 8687  ok
tools  trace -trailing
```

Es erscheint dieses Bild, wenn Sie nach dem Erscheinen einer jeden Zeile solange die <CR>Taste drücken, bis wieder ok erscheint:

```
8658  66A  2DUP              1B 8687
865A  AB2  BOUNDS            1B 8687 1B 8687
```

```
865C  A80  (?DO            8687 86A2 1B 8687
8660  66A  2DUP            1B 8687
8662  6E2  +               1B 8687 1B 8687
8664  789  1-              86A2 1B 8687
8666  469  C@              86A1 1B 8687
8668 2224  BL              20 1B 8687
866A  91B  =               20 20 1B 8687
866C  B34  ?BRANCH         FFFF 1B 8687
8670  CFB  LEAVE           1B 8687
8678  40D  UNNEST          1B 8687
```

Betrachten Sie zuerst die Syntax von **TRACE** :

```
trace <name1>
```

Hierbei ist <name1> das zu tracende Wort, wobei die vom Tracer zurückgelieferte Information so aussieht:

```
addr1 addr2 <name2>  Werte
```

addr1	ist eine Adresse im Parameterfeld von <name1>, nämlich die, in der addr2 steht.
addr2	ist die KompilationsAdresse von <name2>.
<name2>	ist das Wort, das als nächstes ausgeführt werden soll !
Werte	sind die Werte, die gerade auf dem Stack liegen.

Wie deuten Sie nun das eben erhaltene Bild ? In der ersten Zeile, die der Tracer beim Abarbeiten des zu untersuchenden Wortes **-TRAILING** angezeigt hat, finden Sie:

```
8658  66A  2DUP            1B 8687
```

Hier ist 1B 8687 der Stackinhalt, wie er von **TESTSTRING COUNT** geliefert wurde, nämlich Adresse und Länge des Strings TESTSTRING. Natürlich können die Zahlen bei Ihnen anders aussehen, je nachdem, wohin **TESTSTRING** und **-TRAILING** kompiliert wurde.
66A ist die Kompilationsadresse von **2DUP** , 8658 die Position von **2DUP** in **-TRAILING** . Auch diese Adressen können sich bei Ihnen geändert haben! Diese Zahlen werden mit ausgegeben, so daß auch im Falle mehrerer Worte mit gleichem Namen eine Identifizierung möglich ist.

Sehen wir uns die Ausgabe nun etwas genauer an.
Bei den ersten beiden Zeilen wächst der Wert ganz links immer um 2. Es ist der Inhalt des Instructionpointers IP, der immer auf die nächste auszuführende Adresse zeigt. Der Inhalt dieser Adresse ist jeweils eine Kompilationsadresse 66A bei 2DUP usw.. Jede Kompilationsadresse benötigt zwei Bytes, daher muß der IP immer um 2 erhöht werden.

Immer? Nein, denn schon die nächste Zeile zeigt eine Ausnahme.
Das Wort **(?DO** erhöht den IP um 4 ! Woher kommt eigentlich **(?DO** , in der Definition von **-TRAILING** stand doch nur **?DO** . **(?DO** ist ein von **?DO** kompi-

liertes Wort, das zusätzlich zur Kompilationsadresse noch einen 16Bit-Wert benötigt, nämlich für den Sprungoffset hinter LOOP , wenn die Schleife beendet ist. Zwei ähnliche Fälle treten noch auf. Das IF aus dem Quelltext hat ein ?BRANCH kompiliert. Es wird gesprungen, wenn der oberste Stackwert FALSE (=Ø) ist. Auch ?BRANCH benötigt einen zusätzlichen 16Bit-Wert für den Sprungoffset.

Nach LEAVE geht es hinter LOOP weiter, es wird UNNEST ausgeführt, das vom ; in -TRAILING kompiliert wurde und das gleiche wie EXIT bewirkt. Damit ist das Wort -TRAILING auch beendet. Das hier gelistete Wort UNNEST ist nicht zu verwechseln mit dem UNNEST des Tracers, siehe unten.m

Wo liegt nun der Fehler in unserer Definition von -TRAILING ?
Bevor Sie weiterlesen, sollten Sie die Fehlerbeschreibung, den Tracelauf und Ihre Vorstellung von der korrekten Arbeitsweise des Wortes noch einmal unter die Lupe nehmen.

Der Stack ist vor und nach -TRAILING gleich geblieben, die Länge des Strings also nicht verändert worden. Offensichtlich wird die Schleife gleich beim ersten Mal verlassen, obwohl das letzte Zeichen des Textes ein Leerzeichen war. Die Schleife hätte also eigentlich mit dem vorletzten Zeichen weiter machen müssen.
Mit anderen Worten: Die Abbruchbedingung in der Schleife ist falsch! Sie ist genau verkehrt herum gewählt. Ersetzt man = durch = NOT oder - , so funktioniert das Wort korrekt. Überlegen Sie bitte, warum auch - statt = NOT eingesetzt werden kann. Tip: Der IF-Zweig wird nicht ausgeführt, wenn der oberste Stackwert FALSE , also gleich NULL ist.

Der volksFORTH83-Tracer gestattet es, jederzeit Befehle einzu-geben, die vor dem Abarbeiten des nächsten Trace-Kommandos ausgeführt werden. Das System wartet nach einem Step auf eine Eingabe von der Tastatur, bevor der nächste Step ausgeführt wird.
Endet eine Zeile mit einem Leerzeichen vor dem <CR> , so erlaubt der Tracer die Eingabe und Verarbeitung einer vollständigen Kommandozeile. Die Anforderung wird wiederholt, wenn in dieser Zeile ein Eingabe-Fehler auftauchen sollte. Damit ist jetzt ein etwas stressfreieres Ändern des Code bzw. des Stack während des Tracens möglich.
So kann man z. B. Stack-Werte verändern oder das Tracen abbrechen. Ändern Sie probehalber beim nächsten Trace-Lauf von -TRAILING durch Eingabe von NOT das TRUE-Flag ($FFFF) auf dem Stack. bevor ?BRANCH ausgeführt wird und verfolgen Sie den weiteren Trace-Lauf. Sie werden bemerken, daß die LOOP ein zweites Mal durchlaufen wird.

Wollen Sie das Tracen und die weitere Ausführung des getraceten Wortes abbrechen, so geben Sie restart ein. RESTART führt einen Warm-Start des FORTHSystems aus und schaltet den Tracer ab. RESTART ist auch die Katastrophen-Notbremse, die man einsetzt, wenn man sieht, daß das System mit dem nächsten Befehl zwangsläufig im ComputerNirwana entschwinden wird.

17.2 Debug

Beim Tracen mit **TRACE** - nomen est omen - haben wir das fehlerhafte Wort isoliert zusammen mit ausgesuchten Parametern analysiert.

Im Gegensatz dazu bietet sich die Möglichkeit an, ein Wort in seiner Umgebung, also in der Definition, in der es fehlerhaft arbeitet, zu untersuchen. Dazu dient der Befehl **DEBUG** :

```
debug <name>
```

Hierbei ist <name> das zu tracende Wort.

17.2.1 Beispiel: EXAMPLE

Bleiben wir bei unserem Beispiel, geben Sie bitte eine Definition mit **-TRAILING** ein. Dabei bietet sich zunächst die gleiche Sequenz wie oben an. Bitte definieren Sie dieses Beispiel :

```
: example  ( -- )
    teststring count
    -trailing  clearstack  ;
```

Nun aktivieren Sie den Tracer mit :

```
debug -trailing
```

Zunächst geschieht noch gar nichts. **DEBUG** hat nur den Tracer "scharf" gemacht. Rufen Sie nun mit **EXAMPLE** ein Wort auf, das <name> enthält, unterbricht der Tracer die Ausführung, wenn er auf <name> stößt. Es erscheint folgendes, Ihnen schon bekanntes Bild:

```
example

8658  66A  2DUP              1B 8687
865A  AB2  BOUNDS            1B 8687 1B 8687
865C  A80  (?DO              8687 86A2 1B 8687
8660  66A  2DUP              1B 8687
8662  6E2  +                 1B 8687 1B 8687
8664  789  1-                86A2 1B 8687
...
```

Die Parameter sind unter diesen Bedingungen natürlich wieder die, die von **TESTSTRING** und **COUNT** auf den Stack gelegt werden. Interessant ist aber hier die Kontrolle, ob **-TRAILING** vielleicht falsche Parameter übergeben bekommt.

17.2.2 NEST und UNNEST

Nützlich ist auch die Möglichkeit, das Wort, das als nächstes zur Ausführung ansteht, seinerseits zu tracen, bis es ins aufrufende Wort zurückkehrt.

Dafür ist das Wort **nest** vorgesehen. Allerdings beschränkt sich dieses Möglichkeit auf Worte, die in Highlevel-FORTH geschrieben sind. Assemblercode läßt sich damit nicht debuggen und Sie erhalten eine Meldung:

```
<name> can't be debugged
```

Bitte tracen Sie unser Beispiel wieder mit **TRACE EXAMPLE** :

Wenn Sie nun wissen wollen, was **-TRAILING** innerhalb des Beispiels macht, so geben Sie bitte **NEST** ein, wenn **-TRAILING** als nächstes auszuführendes Wort angezeigt wird.

Sie erhalten dann:

```
trace example

86B0 8684 TESTSTRING
86B2  EEF COUNT              8686
86B4 8656 -TRAILING          1B 8687  nest
8658  66A 2DUP               1B 8687
865A  AB2 BOUNDS             1B 8687 1B 8687
865C  A80 ?DO                8687 86A2 1B 8687
8660  66A 2DUP               1B 8687
8662  6E2 +                  1B 8687 1B 8687
8664  789 1-                 86A2 1B 8687
8666  469 C@                 86A1 1B 8687
8668 2224 BL                 20 1B 8687
866A  91B =                  20 20 1B 8687
866C  B34 ?BRANCH            FFFF 1B 8687
8670  CFB LEAVE              1B 8687
8678  40D UNNEST             1B 8687
86B6 162E CLEARSTACK         1B 8687
86B8  40D UNNEST             ok
```

Beachten Sie bitte, daß die Zeilen jetzt eingerückt dargestellt werden, bis der Tracer automatisch in das aufrufende Wort zurückkehrt. Der Gebrauch von **NEST** ist nur dadurch eingeschränkt, daß sich einige Worte, die den ReturnStack manipulieren, mit **NEST** nicht tracen lassen, da der Tracer selbst Gebrauch vom ReturnStack macht. Auf solche Worte muß man den Tracer mit **DEBUG** ansetzen.

Wollen Sie das Tracen eines Wortes beenden, ohne die Ausführung des Wortes abzubrechen, so benutzen Sie **unnest** .

Ist der Tracer geladen, so kommen Sie an das tief im System steckende **UNNEST** , einem Synonym für **EXIT** , das ausschließlich vom ; kompiliert wird, nicht mehr heran und benutzen statt dessen das Tracer-UNNEST, das Sie eine Ebene im Tracelauf zurückbringt.

Manchmal hat man in einem Wort vorkommende Schleifen beim ersten Durchlauf als korrekt erkannt und möchte diese nicht weiter tracen. Das kann sowohl bei **DO...LOOP**s der Fall sein als auch bei Konstruktionen mit **BEGIN...WHILE...REPEAT** oder **BEGIN...UNTIL**. In diesen Fällen gibt man am Ende der Schleife das Wort **endloop** ein. Die Schleife wird dann in Echtzeit abgearbeitet und der Tracer mel-

det sich erst wieder, wenn er nach dem Wort angekommen ist, bei dem **ENDLOOP** eingegeben wurde.
Sollte statt dessen die Meldung: **ENDLOOP COMPILE ONLY** zu sehen sein, so ist das Vokabular **TOOLS** nicht in der Suchreihenfolge . Haben Sie den Fehler gefunden und wollen deshalb nicht mehr tracen, so müssen Sie nach dem Ende des Tracens **END-TRACE** oder jederzeit **RESTART** eingeben, ansonsten bleibt der Tracer "scharf", was zu merkwürdigen Erscheinungen führen kann; außer dem verringert sich bei eingeschaltetem Tracer die Geschwindigkeit des Systems.
Beachten Sie bitte auch, daß Sie für die Worte **DEBUG** und **TRACE** das Vokabular **TOOLS** mit in die Suchreihenfolge aufgenommen haben. Sie sollten also nach hoffentlich erfolgreichem Tracelauf die Suchordnung wieder umschalten, weil der Befehl **RESTART** zwar den Tracer abschaltet, aber die Suchreihenfolge nicht zurückschaltet.

Wenn man sich eingearbeitet hat, ist der Tracer ein wirklich verblüffendes Werkzeug, mit dem man sehr viele Fehler schnell finden kann. Er ist gleichsam ein Mikroskop, mit dem man sehr tief ins Innere von FORTH schauen kann.

17.3 Stacksicherheit

Anfänger neigen häufig dazu, Fehler bei der Stackmanipulation zu machen. Erschwerend kommt häufig hinzu, daß sie viel zu lange Worte schreiben, in denen es dann von unübersichtlichen Stackmanipulationen nur so wimmelt. Es gibt einige Worte, die sehr einfach sind und Fehler bei der Stackmanipulation früh erkennen helfen. Denn leider führen schwerwiegende Stackfehler zu "mysteriösen" Systemcrashs.

In Schleifen führt ein nicht ausgeglichener Stack oft zu solchen Fehlern. Während der Testphase eines Programms oder Wortes sollte man daher bei jedem Schleifendurchlauf prüfen, ob der Stack evtl. über- oder leerläuft. Das geschieht durch Eintippen von :

```
: LOOP    compile ?stack [compile] LOOP   ; immediate restrict
: +LOOP   compile ?stack [compile] +LOOP  ; immediate restrict
: UNTIL   compile ?stack [compile] UNTIL  ; immediate restrict
: REPEAT  compile ?stack [compile] REPEAT ; immediate restrict
: :  : compile ?stack ;
```

Versuchen Sie ruhig, herauszufinden wie die letzte Definition funktioniert. Es ist nicht kompliziert. Durch diese Worte bekommt man sehr schnell mitgeteilt, wann ein Fehler auftrat. Es erscheint dann die Fehlermeldung

```
<name> stack full
```

, wobei <name> der zuletzt vom Terminal eingegebene Name ist.Wenn man nun überhaupt keine Ahnung hat, wo der Fehler auftrat, so gebe man ein:

```
: unravel
    rdrop rdrop rdrop    \ delete errorhandlernest
    cr ." trace dump on abort is :" cr
```

```
        BEGIN rp@ r0 @ -      \ until stack empty
        WHILE r> dup 8 u.r space
              2- @ >name  .name cr
        REPEAT (error ;

    ' unravel  errorhandler !
```

Sie bekommen dann bei Eingabe von 1 2 0 */ ungefähr folgenden Ausdruck :

```
trace dump on abort is:
E32  */MOD 1D8A EXECUTE
1E24 PARSER
20CD INTERPRET
20FA 'QUIT / division overflow
```

'QUIT **INTERPRET** **PARSER** und **EXECUTE** rühren vom Textinterpreter her. Interessant wird es bei ***/MOD** . Wir wissen, daß ***/MOD** von ***/** aufgerufen wird. ***/MOD** ruft nun wieder **M/MOD** auf, in **M/MOD** gehts weiter nach **UM/MOD** . Dieses Wort ist in Code geschrieben und "verursachte" den Fehler, indem es eine Division durch Null ausführte.

Nicht immer treten Fehler in Schleifen auf. Es kann auch der Fall sein, daß ein Wort zu wenig Argumente auf dem Stack vorfindet, weniger nämlich, als Sie für dieses Wort vorgesehen haben. Diesen Fall sichert **ARGUMENTS** . Die Definition dieses Wortes ist:

```
: arguments   ( n --)
    depth 1- < abort" not enough arguments" ;
```

Es wird folgendermaßen benutzt:

```
: -trailing ( addr len)  2 arguments ... ;
```

wobei die drei Punkte den Rest des Quelltextes andeuten sollen.

Findet **-TRAILING** nun weniger als zwei Werte auf dem Stack vor, so wird eine Fehlermeldung ausgegeben. Natürlich kann man damit nicht prüfen, ob die Werte auf dem Stack wirklich für **-TRAILING** bestimmt waren.

Sind Sie als Programmierer sicher, daß an einer bestimmten Stelle im Programm eine bestimmte Anzahl von Werten auf dem Stack liegt, so können Sie das ebenfalls sicherstellen:

```
: isdepth   ( n --)  depth 1-  - abort" wrong depth" ;
```

ISDEPTH bricht das Programm ab, wenn die Zahl der Werte auf dem Stack nicht gleich n ist, wobei n natürlich nicht mitgezählt wird. Es wird analog zu **ARGUMENTS** benutzt. Mit diesen Worten lassen sich Stackfehler oft feststellen und lokalisieren.

17.4 Aufrufgeschichte

Möchte man wissen, was geschah, bevor ein Fehler auftrat und nicht nur, wo er auftrat.- denn nur diese Information liefert **UNRAVEL** - so kann man einen modifizierten Tracer verwenden, bei dem man nicht nach jeder Zeile <CR> drücken muß:

```
: :  ( -- )
     :
     Does>  cr  rdepth 2* spaces
            dup 2- >name .name  >r ;
```

Hierbei wird ein neues Wort mit dem Namen : definiert, das zunächst das alte Wort : aufruft und so ein Wort im Dictionary erzeugt. Das Laufzeitverhalten dieses Wortes wird aber so geändert, daß es sich jedesmal wieder ausdruckt, wenn es aufgerufen wird. Alle Worte, die nach der Redefinition, so nennt man das erneute Definieren eines schon bekannten Wortes, des : definiert wurden, weisen dieses Verhalten auf.

Beispiel:

```
: /  / ;
: rechne   ( -- n)  1 2 3 / / ;
RECHNE / / RECHNE division overflow
```

Wir sehen also, daß erst bei der zweiten Division der Fehler auftrat. Das ist auch logisch, denn 2 3 / ergibt Ø .

Sie sind sicher in der Lage, die Grundidee dieses zweiten Tracers zu verfeinern. Ideen wären z.B.:
- Ausgabe der Werte auf dem Stack bei Aufruf eines Wortes
- Die Möglichkeit, Teile eines Tracelaufs komfortabel zu unterdrücken.

17.5 Dump

Einen Speicherdump benötigt man beim Programmieren sehr oft, mindestens dann, wenn man eigene Datenstrukturen anschauen will. Oft ist es dann hinderlich, eigene Worte zur womöglich gar formatierten Ausgabe der Datenstrukturen schreiben zu müssen. In diesen Fällen benötigt man ein Wort, das einen Speicherdump ausgibt. Das volksFORTH besitzt zwei Worte zum Dumpen von Speicherblöcken sowie einen Dekompiler, der auch für Datenstrukturen verwendet werden kann.

dump (addr n --)

Ab addr werden n Bytes formatiert ausgegeben. Dabei steht am Anfang einer Zeile die Adresse, dann folgen 16 Byte, die in der Mitte zur besseren Übersicht getrennt sind und dann die Ascii-Darstellung. Dabei werden nur Zeichen im Bereich zwischen $2Ø und $7F ausgegeben. Die Ausgabe läßt sich jederzeit mit einer beliebigen Taste unterbrechen oder mit <Esc> abbrechen.
Mit PRINT läßt sich die Ausgabe auf einen Drucker umleiten.
Beispiel: print pad 4Ø dump display
Das abschließende DISPLAY sorgt dafür, daß wieder der Bild-schirm als Ausgabegerät gesetzt wird.

Möchte man Speicherbereich außerhalb des FORTH-Systems dumpen, gibt es dafür das Wort:

ldump (laddr n --)

erwartet eine - doppelt lange - absolute Speicheradresse und wie oben die Anzahl der auszugebenden Bytes. Die Ausgabe auf einen Drucker geschieht genau so wie oben beschrieben.

17.6 Dekompiler

Ein Dekompiler gehört so zu sagen zum guten Ton eines FORTH Systems, war er bisher doch die einzige Möglichkeit, wenigstens ungefähr den Aufbau eines Systems zu durchschauen. Bei volks FORTH83 ist das anders, und zwar aus zwei Gründen: Sie haben sämtliche Quelltexte vorliegen, und es gibt die VIEW-Funktion. Letztere ist normalerweise sinn voller als der beste Dekompiler, da kein Dekompiler in der Lage ist, z.B. Stackkommentare zu rekonstruieren.

Der Tracer ist beim Debugging sehr viel hilfreicher als ein Dekompiler, da er auch die Verarbeitung von Stackwerten erkennen läßt. Damit sind Fehler leichter aufzufinden.
Dennoch gibt es natürlich auch im volksFORTH einen Dekompiler, zum einen als zuladbares Werkzeug und zum anderen in einfacher von Hand zu bedienender Form.

Der zuladbare Dekompiler SEE wird mit

```
include see.scr
```

geladen und in der Form

```
see <name>
```

benutzt. Daraufhin wird das Wort dekompilert.

Als ständig verfügbares Dekompiler-Werkzeug stehen im Vokabular TOOLS folgende Worte zur Verfügung:

N name (addr -- addr')
druckt den Namen des bei addr kompilierten Wortes aus und setzt addr auf das nächste Wort.

L literal (addr -- addr')
wird nach LIT benutzt und druckt den Inhalt von addr als Zahl aus. Es wird also nicht versucht, den Inhalt, wie bei N, als FORTH-Wort zu interpretieren.

S string (addr -- addr')
wird nach ABORT" , " , ." und allen anderen Worten benutzt, auf die ein String folgt. Der String wird ausgedruckt und addr entsprechend erhöht, sodaß sie hinter den String zeigt.

C character (addr -- addr')
druckt den Inhalt von addr als Ascii-Zeichen aus und geht ein Byte weiter. Damit kann man eigene Datenstrukturen ansehen.

B branch (addr -- addr')
wird nach BRANCH oder ?BRANCH benutzt und druckt den Inhalt einer Adresse als Sprungoffset und Sprungziel aus.

D dump (addr n -- addr')
dumped n Bytes. Wird benutzt, um selbstdefinierte Daten strukturen anzusehen.
Siehe auch DUMP und LDUMP .

Sehen wir uns nun ein Beispiel zur Benutzung des Dekompilers an. Geben Sie bitte folgende Definition ein:

```
: test  ( n --)
12 = IF  cr ." Die Zahl ist zwölf !"  THEN ;
```

Rufen Sie das Vokabular TOOLS durch Nennen seines Namens auf und ermitteln Sie die Adresse des ersten in TEST kompilierten Wortes:

```
' test >body
```

Jetzt können Sie TEST nach folgendem Muster dekompilieren:

```
n 80B0:  856  CLIT
c 80B2:   12
n 80B3:  92A  =
n 80B5:  B43  ?BRANCH
b 80B7:   1B  80D2
n 80B9: 2BE9  CR
n 80BB: 127E  (."
s 80BD:   14  Die Zahl ist zwölf !
n 80D2:  41C  UNNEST
```

Die erste Adresse ist die, an der im Wort **TEST** die anderen Worte kompiliert sind. Die zweite ist jeweils die Kompilationsadresse der Worte, danach folgen die sonstigen Ausgaben des Dekompilers.
Probieren Sie dieses Beispiel auch mit dem Tracer aus: **20 trace test** und achten Sie auf die Unterschiede. Sie werden sehen, daß der Tracer aussagefähiger und dazu noch einfacher zu bedienen ist.

Wenn Sie sich die Ratschläge und Tips zu Herzen genommen haben und noch etwas den Umgang mit den Hilfsmitteln üben, werden Sie sich sicher nicht mehr vorstellen können, wie Sie jemals in anderen Sprachen ohne diese Hilfsmittel ausgekommen sind. Bedenken Sie bitte auch: Diese Mittel sind kein Heiligtum - oft wird Sie eine Modifikation schneller zum Ziel führen. Scheuen Sie sich nicht, sich vor dem Bearbeiten einer umfangreichen Aufgabe erst die geeigneten Hilfsmittel zu programmieren.

17.7 Glossar

restart (--)
ermöglicht gegenüber den Versionen auf dem ATARI und dem C64 auf Grund der Konstruktion ausschließlich den Ausstieg aus dem Single-Step-Trace-Modus .

end-trace (--)
dient lediglich dazu, um einen **DEBUG**-Befehl rückgängig zu machen. Schaltet den Tracer ab, der durch "patchen" der Next-Routine arbeitet. Die Ausführung des aufrufenden Wortes wird fortgesetzt.

next-link (-- addr)
addr ist die Adresse einer Uservariablen, die auf die Liste aller Nextroutinen zeigt. Der Assembler erzeugt bei Verwendung des Wortes **NEXT** Code, für den ein Zeiger in diese Liste eingetragen wird. Die so entstandene Liste wird vom Tracer benutzt, um alle im System vorhandenen Kopien des Codes für **NEXT** zu modifizieren.

18. Begriffe

18.1 Entscheidungskriterien

Bei Konflikten läßt sich das Standardteam von folgenden Kriterien in Reihenfolge ihrer Wichtigkeit leiten:

1. Korrekte Funktion - bekannte Einschränkungen, Eindeutigkeit.
2. Transportabilität - wiederholbare Ergebnisse, wenn Programme zwischen Standardsystemen portiert werden.
3. Einfachheit
4. Klare, eindeutige Namen - die Benutzung beschreiben - der statt funktionaler Namen, z.B. **[COMPILE]** statt **'c**, und **ALLOT** statt **dp+!** .
5. Allgemeinheit
6. Ausführungsgeschwindigkeit
7. Kompaktheit
8. Kompilationsgeschwindigkeit
9. Historische Kontinuität
10. Aussprechbarkeit
11. Verständlichkeit - es muß einfach gelehrt werden können

18.2 Definition der Begriffe

Es werden im allgemeinen die amerikanischen Begriffe beibehalten, es sei denn, der Begriff ist bereits im Deutschen geläufig. Wird ein deutscher Begriff verwendet, so wird in Klammern der engl. Originalbegriff beigefügt; wird der Originalbegriff beibehalten, so wird in Klammern eine möglichst treffende Übersetzung angegeben.

Adresse, Byte (address, byte)
: Eine 16bit Zahl ohne Vorzeichen, die den Ort eines 8bit Bytes im Bereich <Ø...65,535> angibt. Adressen werden wie Zahlen ohne Vorzeichen manipuliert.
Siehe: "Arithmetik, 2er-komplement"

Adresse, Kompilation (address, compilation)
: Der Zahlenwert, der zur Identifikation eines FORTH Wortes kompiliert wird. Der Adressinterpreter benutzt diesen Wert, um den zu jedem Wort gehörigen Maschinencode aufzufinden.

Adresse, Natürliche (address, native machine)
Die vorgegebene Adressdarstellung der Computerhardware.

Adresse, Parameterfeld (address, parameter field) "pfa"
Die Adresse des ersten Bytes jedes Wortes, das für das Ablegen von Kompilationsadressen (bei :-definitionen) oder numerischen Daten bzw. Textstrings usw. benutzt wird.

anzeigen (display)
Der Prozess, ein oder mehrere Zeichen zum aktuellen Ausgabegerät zu senden. Diese Zeichen werden normalerweise auf einem Monitor angezeigt bzw. auf einem Drucker gedruckt.

Arithmetik, 2er-komplement (arithmetic, two's complement)
Die Arithmetik arbeitet mit Zahlen in 2er-Komplementdarstellung; diese Zahlen sind, je nach Operation, 16bit oder 32bit weit. Addition und Subtraktion von 2er-Komplementzahlen ignorieren Überlaufsituationen. Dadurch ist es möglich, daß die gleichen Operatoren benutzt werden können, gleichgültig, ob man die Zahl mit Vorzeichen (-32,768...32,767 bei 16-Bit) oder ohne Vorzeichen (Ø...65,535 bei 16-Bit) benutzt.

Block (block)
Die 1Ø24 Byte Daten des Massenspeichers, auf die über Blocknummern im Bereich Ø...Anzahl_existenter_Blöcke-1 zugegriffen wird. Die exakte Anzahl der Bytes, die je Zugriff auf den Massenspeicher übertragen werden, und die Übersetzung von Blocknummern in die zugehörige Adresse des Laufwerks und des physikalischen Satzes, sind rechnerabhängig.
Siehe: "Blockpuffer" und "Massenspeicher"

Blockpuffer (block buffer)
Ein 1Ø24 Byte langer Hauptspeicherbereich, in dem ein Block vorübergehend benutzbar ist. Ein Block ist in höchstens einem Blockpuffer enthalten.

Byte (byte)
Eine Einheit von 8bit. Bei Speichern ist es die Speicherkapazität von 8bits.

Kompilation (compilation)

Der Prozess, den Quelltext in eine interne Form umzuwandeln, die später ausgeführt werden kann. Wenn sich das System im Kompilationszustand befindet, werden die Kompilationsadressen von Worten im Dictionary abgelegt, so daß sie später vom Adresseninterpreter ausgeführt werden können. Zahlen werden so kompiliert, daß sie bei Ausführung auf den Stack gelegt werden. Zahlen werden aus dem Quelltext ohne oder mit negativem Vorzeichen akzeptiert und gemäß dem Wert von **BASE** umgewandelt.

Siehe: "Zahl", "Zahlenumwandlung", "Interpreter, Text" und "Zustand"

Definition (Definition)

Siehe: "Wortdefinition"

Dictionary (Wörterbuch)

Eine Struktur von Wortdefinitionen, die im Hauptspeicher des Rechners angelegt ist. Sie ist erweiterbar und wächst in Richtung höherer Speicheradressen. Einträge sind in Vokabularen organisiert, so daß die Benutzung von Synonymen möglich ist, d.h. gleiche Namen können, in verschiedenen Vokabularen enthalten, vollkommen verschiedene Funktionen auslösen.

Siehe: "Suchreihenfolge"

Division, floored (division, floored)

Ganzzahlige Division, bei der der Rest das gleiche Vorzeichen hat wie der Divisor oder gleich Null ist; der Quotient wird gegen die nächstkleinere ganze Zahl gerundet. Bemerkung: Ausgenommen von Fehlern durch Überlauf gilt: `N1 N2 SWAP OVER /MOD ROT * +` ist identisch mit `N1` .

Siehe: "floor, arithmetisch"

Beispiele: Dividend	Divisor	Rest	Quotient
10	7	3	1
-10	7	4	-2
10	-7	-4	-2
-10	-7	-3	1

Empfangen receive)

Der Prozess, der darin besteht, ein Zeichen von der aktuellen Eingabeeinheit zu empfangen. Die Anwahl einer Einheit ist rechnerabhängig.

Falsch (false)

Die Zahl Null repräsentiert den "Falschzustand" eines Flags.

Fehlerbedingung (error condition)

Eine Ausnahmesituation, in der ein Systemverhalten erfolgt, das nicht mit der erwarteten Funktion übereinstimmt. Im der Beschreibung der einzelnen Worte sind die möglichen Fehlerbedingungen und das dazugehörige Systemverhalten beschrieben.

Flag (logischer Wert)

Eine Zahl, die eine von zwei möglichen Werten hat, falsch oder wahr.
Siehe: "Falsch" , "Wahr"

Floor, arithmetic

Z sei eine reelle Zahl. Dann ist der Floor von Z die größte ganze Zahl, die kleiner oder gleich Z ist.

Der Floor von +Ø,6 ist Ø
Der Floor von -Ø,4 ist -1

Glossar (glossary)

Eine umgangssprachliche Beschreibung, die die zu einer Wortdefinition gehörende Aktion des Computers beschreibt - die Beschreibung der Semantik des Wortes.

Interpreter, Adressen (interpreter, address)

Die Maschinencodeinstruktionen, die die kompilierten Wort-definitionen ausführen, die aus Kompilationsadressen bestehen.

Interpreter, Text (interpreter, text)

Eine Wortdefinition, die immer wieder einen Wortnamen aus dem Quelltext holt, die zugehörige Kompilationsadresse bestimmt und diese durch den Adressinterpreter ausführen läßt. Quelltext, der als Zahl interpretiert wird, hinter-läßt den entsprechenden Wert auf dem Stack.
Siehe: "Zahlenumwandlung"

Kontrollstrukturen (structure, control)

Eine Gruppe von Worten, die, wenn sie ausgeführt werden, den Programmfluß verändern. Beispiele von Kontrollstrukturen sind:

DO ... LOOP
BEGIN ... WHILE ... REPEAT
IF ... ELSE ... THEN

laden (load)

Das Umschalten des Quelltextes zum Massenspeicher. Dies ist die übliche Methode, dem Dictionary neue Definitionen hinzuzufügen.

Massenspeicher (mass storage)

Speicher, der ausserhalb des durch FORTH adressierbaren Bereiches liegen kann. Auf den Massenspeicher wird in Form von 1Ø24 Byte großen Blökken zugegriffen. Auf einen Block kann innerhalb des FORTH-Adressbereichs in einem Blockpuffer zugegriffen werden. Wenn ein Block als verändert (**UPDATE**) gekennzeichnet ist, wird er letztendlich wieder auf den Massenspeicher zurückgeschrieben.

Programm (program)

Eine vollständige Ablaufbeschreibung in FORTH-Quelltext, um eine bestimmte Funktion zu realisieren.

Quelltext (input stream)

Eine Folge von Zeichen, die dem System zur Bearbeitung durch den Textinterpreter zugeführt wird. Der Quelltext kommt üblicherweise von der aktuellen Eingabeeinheit (über den Texteingabepuffer) oder dem Massenspeicher (über einen Blockpuffer). **BLK** , **>IN** , **TIB** und **#TIB** charakterisieren den Quelltext. Worte, die **BLK** , **>IN** , **TIB** oder **#TIB** benutzen und/oder verändern, sind dafür verantwortlich, die Kontrolle des Quelltextes aufrechtzuerhalten oder wiederherzustellen. Der Quelltext reicht von der Stelle, die durch den Relativzeiger **>IN** angegeben wird, bis zum Ende. Wenn **BLK** Null ist, so befindet sich der Quelltext an der Stelle, die durch **TIB** adressiert wird, und er ist **#TIB** Bytes lang. Wenn **BLK** ungleich Null ist, so ist der Quelltext der Inhalt des Blockpuffers, der durch BLK angegeben ist, und er ist 1Ø24 Byte lang.

Rekursion (recursion)

Der Prozess der direkten oder indirekten Selbstreferenz.

Screen (Bildschirm)

Ein Screen sind Textdaten, die zum Editieren aufbereitet sind. Nach Konvention besteht ein Screen aus 16 Zeilen zu je 64 Zeichen. Die Zeilen werden von Ø bis 15 durchnumeriert. Screens enthalten normalerweise Quelltext, können jedoch auch dazu benutzt werden, um Massenspeicherdaten zu betrachten. Das erste Byte eines Screens ist gleichzeitig das erste Byte eines Massenspeicherblocks; dies ist auch der Anfangspunkt für Quelltextinterpretation während des Ladens eines Blocks.

Suchreihenfolge (search order)

Eine Spezifikation der Reihenfolge, in der ausgewählte Vokabulare im Dictionary durchsucht werden. Die Suchreihenfolge besteht aus einem auswechselbaren und einem festen Teil, wobei der auswechselbare Teil immer als erstes durchsucht wird. Die Ausführung eines Vokabularnamens macht es zum ersten Vokabular in der Suchreihenfolge, wobei das Vokabular, das vorher als erstes durchsucht worden war, verdrängt wird. Auf dieses erste Vokabular folgt, soweit spezifiziert, der feste Teil der Suchreihen-folge, der danach durchsucht wird. Die Ausführung von ALSO übernimmt das Vokabular im auswechselbaren Teil in den festen Teil der Suchreihenfolge. Das Dictionary wird immer dann durchsucht, wenn ein Wort durch seinen Namen aufge-funden werden soll.

stack, data (Datenstapel)

Eine "Zuletzt-rein, Zuerst-raus" (last-in, first-out) Struktur, die aus einzelnen 16bit Daten besteht. Dieser Stack wird hauptsächlich zum Ablegen von Zwischenergebnissen während des Ausführens von Wortdefinitionen benutzt. Daten auf dem Stack können Zahlen, Zeichen, Adressen, Boole'sche Werte usw. sein. Wenn der Begriff "Stapel" oder "Stack" ohne Zusatz benutzt wird, so ist immer der Datenstack gemeint.

stack, return (Rücksprungstapel)

Eine "Zuletzt-rein, Zuerst-raus" Struktur, die hauptsächlich Adressen von Wortdefinitionen enthält, deren Ausführung durch den Adressinterpreter noch nicht beendet ist. Wenn eine Wortdefinition eine andere Wortdefinition aufruft, so wird die Rücksprungadresse auf dem Returnstack abgelegt. Der Returnstack kann zeitweise auch für die Ablage anderer Daten benutzt werden.

String, counted (abgezählte Zeichenkette)

Eine Hintereinanderfolge von 8bit Daten, die im Speicher durch ihre niedrigste Adresse charakterisiert wird. Das Byte an dieser Adresse enthält einen Zahlenwert im Bereich <Ø...255>, der die Anzahl der zu diesem String gehörigen Bytes angibt, die unmittelbar auf das Countbyte folgen. Die Anzahl beinhaltet nicht das Countbyte selber. Counted Strings enthalten normalerweise ASCII-Zeichen.

String, Text (Zeichenkette)

Eine Hintereinanderfolge von 8bit Daten, die im Speicher durch ihre niedrigste Adresse und ihre Länge in Bytes charakterisiert ist. Strings enthalten normalerweise ASCII-Zeichen. Wenn der Begriff "String" alleine oder in Verbindung mit anderen Begriffen benutzt wird, so sind Textstrings gemeint.

Userarea (Benutzerbereich)
Ein Gebiet im Speicher, das zum Ablegen der Uservariablen benutzt wird und für jeden einzelnen Prozeß/Benutzer getrennt vorhanden ist.

Uservariable (Benutzervariable)
Eine Variable, deren Datenbereich sich in der Userarea befindet. Einige Systemvariablen werden in der Userarea gehalten, sodaß die Worte, die diese benutzen, für mehrere Prozesse/Benutzer gleichzeitig verwendbar sind.

Vokabular (vocabulary)
Eine geordnete Liste von Wortdefinitionen. Vokabulare werden vorteilhaft benutzt, um Worte voneinander zu unterscheiden, die gleiche Namen haben (Synonyme). In einem Vokabular können mehrere Definitionen mit dem gleichen Namen existieren; diesen Vorgang nennt man redefinieren. Wird das Vokabular nach einem Namen durchsucht, so wird die jüngste Redefinition gefunden.

Vokabular, Kompilation (vocabulary, compilation)
Das Vokabular, in das neue Wortdefinitionen eingetragen werden.

Wahr (true)
Ein Wert, der nicht Null ist, wird als "wahr" interpre-tiert. Wahrheits-werte, die von Standard-FORTH-Worten errechnet werden, sind 16bit Zahlen, bei denen alle 16 Stellen auf "1" gesetzt sind, so daß diese zum Maskieren benutzt werden können.

Wort, Definierendes (defining word)
Ein Wort, das bei Ausführung einen neuen Dictionary-Eintrag im Kompi-lationsvokabular erzeugt. Der Name des neuen Wortes wird dem Quelltext entnommen. Wenn der Quelltext erschöpft ist, bevor der neue Name er-zeugt wurde, so wird die Fehlermeldung "ungültiger Name" ausgegeben. Beispiele von definierenden Worten sind: : **CONSTANT CREATE**

Wort, immediate (immediate word)
Ein Wort, das ausgeführt wird, wenn es während der Kompilation oder Interpretation aufgefunden wird. Immediate Worte behandeln Sondersitua-tionen während der Kompilation.
Siehe z.B. **IF LITERAL ."** usw.

Wortdefinition (word definition)

Eine mit einem Namen versehene, ausführbare FORTH-Proze-dur, die ins Dictionary kompiliert wurde. Sie kann durch Maschinencode, als eine Folge von Kompilationsadressen oder durch sonstige kompilierte Worte spezifiziert sein. Wenn der Begriff "Wort" ohne Zusatz benutzt wird, so ist im allgemeinen eine Wortdefinition gemeint.

Wortname (word name)

Der Name einer Wortdefinition. Wortnamen sind maximal 31 Zeichen lang und enthalten kein Leerzeichen. Haben zwei Definitionen verschiedene Namen innerhalb desselben Voka-bulars, so sind sie eindeutig auffindbar, wenn das Vokabular durchsucht wird.
Siehe: "Vokabular"

Zahl (number)

Wenn Werte innerhalb eines größeren Feldes existieren, so sind die höherwertigen Bits auf Null gesetzt. 16bit Zahlen sind im Speicher so abgelegt, dass sie in zwei benachbar-ten Byteadressen enthalten sind. Die Bytereihenfolge ist rechnerabhängig. Doppeltgenaue Zahlen (32bit) werden auf dem Stack so abgelegt, daß die höherwertigen 16bit (mit dem Vorzeichenbit) oben liegen. Die Adresse der niederwer-tigen 16bit ist um zwei größer als die Adresse der höherwertigen 16bit, wenn die Zahl im Speicher abgelegt ist.
Siehe: "Arithmetik, 2er-komplement" und "Zahlentypen"

Zahlenausgabe, bildhaft (pictured numeric output)

Durch die Benutzung elementarer Worte für die Zahlenaus-gabe (z.B. **<# # #s #>**) werden Zahlenwerte in Textstrings umgewandelt. Diese Definitionen werden in einer Folge benutzt, die ein symbolisches Bild des gewünschten Ausga-beformates darstellen. Die Umwandlung schreitet von der niedrigstwertigen zur höchstwertigen Ziffer fort und die umgewandelten Zeichen werden von höheren gegen niedrigere Speicheradressen abgelegt.

Zahlenausgabe, freiformatiert (free field format)

Zahlen werden in Abhängigkeit von **BASE** umgewandelt und ohne führende Nullen, aber mit einem folgenden Leerzei-chen, angezeigt. Die Anzahl von Stellen, die angezeigt werden, ist die Minimalanzahl von Stellen - mindestens eine - die notwendig sind, um die Zahl eindeutig darzustellen.
Siehe: "Zahlenumwandlung"

Zahlentypen (number types)

Alle Zahlentypen bestehen aus einer spezifischen Anzahl von Bits. Zahlen mit oder ohne Vorzeichen bestehen aus bewerteten Bits. Bewertete Bits innerhalb einer Zahl haben den Zahlenwert einer Zweierpotenz, wobei das am weitesten rechts stehende Bit (das niedrigstwertige) einen Wert von zwei hoch null hat. Diese Bewertung setzt sich bis zum am weitesten links stehenden Bit fort, wobei sich die Bewertung für jedes Bit um eine Zweierpotenz erhöht. Für eine Zahl ohne Vorzeichen ist das am weitesten links stehende Bit in diese Bewertung eingeschlossen, sodaß für eine solche 16bit Zahl das ganz linke Bit den Wert 32768 hat. Für Zahlen mit Vorzeichen wird die Bewertung des ganz linken Bits negiert, sodaß es bei einer 16bit Zahl den Wert -32768 hat. Diese Art der Wertung für Zahlen mit Vorzeichen wird 2er-komplementdarstellung genannt.Nicht spezifizierte, bewertete Zahlen sind mit oder ohne Vorzeichen; der Programmkontext bestimmt, wie die Zahl zu interpretieren ist.

Zahlenumwandlung (number conversion)

Zahlen werden intern als Binärzahlen geführt und extern durch graphische Zeichen des ASCII Zeichensatzes darge-stellt. Die Umwandlung zwischen der internen und externen Form wird unter Beachtung des Wertes von **BASE** durchge-führt, um die Ziffern einer Zahl zu bestimmen. Eine Ziffer liegt im Bereich von Null bis **BASE**-1. Die Ziffer mit dem Wert Null wird durch das ASCII-zeichen "Ø" (Position 3/Ø, dezimalwert 48) dargestellt. Diese Zifferndarstellung geht den ASCII-code weiter aufwärts bis zum Zeichen "9", das dem dezimalen Wert neun entspricht. Werte, die jenseits von neun liegen, werden durch die ASCII-zeichen beginnend mit "A", entsprechend dem Wert zehn usw. bis zum ASCII-zeichen "~", entsprechend einundsiebzig, dargestellt.Bei einer negativen Zahl wird das ASCII-Zeichen "-" den Ziffern vorangestellt. Bei der Zahleneingabe kann der aktuelle Wert von **BASE** für die gerade umzuwandelnde Zahl dadurch umgangen werden, daß den Ziffern ein "Zahlenbasisprefix" vorangestellt wird. Dabei wird durch das Zeichen "%" die Basis vorübergehend auf den Wert zwei gesetzt, durch "&" auf den Wert zehn und durch "$" oder "h" auf den Wert sechzehn. Bei negativen Zahlen folgt das Zahlenbasisprefix dem Minuszeichen. Enthält die Zahl ein Komma oder einen Punkt, so wird sie als 32bit Zahl umgewandelt.

Zeichen (character)

Eine 8bit Zahl, deren Bedeutung durch den ASCII-Standard festgelegt ist. Wenn es in einem größeren Feld gespeichert ist, so sind die höherwertigen Bits auf Null gesetzt.

Zustand (mode)

Der Textinterpreter kann sich in zwei möglichen Zuständen befinden: dem interpretierenden oder dem kompilierenden Zustand. Die Variable STATUS wird vom System entsprechend gesetzt, und zwar enthält sie bei der Interpretation eine False-flag, bei der Kompilation eine True-flag.
Siehe: "Interpreter, Text" und "Kompilation"

Indexverzeichnis

. ..... 110, 140
! 108
," 76, 155
"lit 76
..... 83
#> 83
#bel 62
#bs 62
#cr 62
#drives 106
#esc 62
#lf 62
#s 83
#tib 67
' 125, 140, 154
'abort 46, 130
'cold 130
'name 141
'quit 130
'restart 130
(..... 76, 150
(at 65
(at? 65
(block 96
(buffer 95
(core? 97
(cr 64
(decode 69
(del 64
(diskerror 106
(emit 73
(error 47
(expect 69
(find 79
(forget 142
(fsearch 104
(key 69
(key? 68
(more 94
(page 64
(quit 130
(r/w 96
(type 73
) 76
* 31
*/ 31
*/mod 31
*block 96
+ 30
+! 108

+load 151
+LOOP 43
+print 66
+thru 151
- 30
--> 151
-1 29
-roll 37
-rot 37
-trailing 77
. 66
." 75
.(....71, 76
.ELSE 41, 42
.file 91
.fname 99
.IF 41, 42
.name 59, 141
.r 66
.s 38
.status 99, 131
.THEN 41, 42
/ 31
/block 96
/drive 105
/mod 31
/string 77
: 6, 114, 182
:Does> 55, 57, 156
; 6, 55, 115
;c: 161
;code 161
< 32
= 32
=or 50
> 32
>asciz 80, 103
>body 143
>drive 105
>expect 70, 78, 82
>in 68, 150
>label 161
>link 143
>mark 51
>name 129, 143
>r 38
>resolve 51
>tib 68
>type 78, 169
? 107
?" 77
?[....162
?[[....162
?cr 65
?diskerror 105
?DO 43

?dup 37
?exit 41
?file 93
?head 145
?pairs 47, 51
@ 107
[....55, 115, 155
['] 154
[[....162
[compile] 155
[FCB] 91
[Method]: 121
\ 151
\\ 151
\needs 151
] 55, 115, 155
]? 162
][....162
]] 162
]]? 162
| 144
~close 104
~creat 103
~dir 104
~disk? 104
~first 105
~next 105
~open 103
~read 104
~select 104
~unlink 104
0 29
0-terminated 75
0< 32
0<> 32
0= 32
0=exit 41
0> 32
1 29
1+ 29
1- 29
2 29
2! 108
2* 30
2+ 29
2- 29
2/ 30
2@ 108
2Constant 114
2drop 36
2dup 37
2over 37
2swap 37
2Variable 114
3 29
3+ 30

4 29
Abbruchbedingung 44
abort 46
abort" 46
abs 30
accumulate 81
activate 165, 169
address 186
Adressinterpreter 146
Adreßraum 107
AGAIN 45
Alias 42, 116, 125, 129
align 109, 141
all-buffers 98
allot 110, 140
allotbuffer 98
also 115, 134
and 33
append 78, 120
Applikation 88
area 63
areakol 63
arguments 181
ASCII-Code 69
asciz 80, 103
Assembler 134, 144, 160
assign 93
Associative: 56
at 65
attach 78
attribut 104
b/blk 94
b/buf 94
b/seg 111
Backspace 62
Batch-File 107
bedingte Kompilierung 41
Befehl 6
BEGIN 44
BELL 62
Betriebssystem 153
between 49
Bildschirm 63, 73
Bildschirmausgabe 64
Bildschirmseite 63
bl 62
blank 109
blk 93, 150
blk/drv 94
block 95, 136, 173, 187
block buffer 187
Blockfeld 136
Block 0 71
bounds 44, 78
buffer 95
bye 152

byte 187
Byteswap 109
c, ... 110, 140
c! 108
c/col 62
c/dis 62
c/l 62
c/row 62
c@ 108
call 88, 107
capacities 92
capacity 95, 105
capital 76
capitalize 77
caps 76
Case: 56
case? 33, 45, 49
catt 64
cd 85, 90
cells 57
cfa 136
character 194
charout 73
clear 144, 145
clearstack 38
close 101
cls 19, 57, 63
cmove 108
cmove> 108
code 136, 138, 160
Codefeld 138, 139
col 65
cold 152
Colon-Definition 147
Colondefinition 41
compilation 188, 192
compile 154
Constant 113
context 133
convert 81
convey 100
copy 99
core? 97
count 78, 110, 136
Count-Byte 75
counted 80, 103
counted String 78, 120
counted strings 148
Countfeld 137
cr 62, 64, 112
Create 55, 113
Create: 55, 57
cswap 109
ctoggle 109
curl 63
curat? 63

curoff 65
curon 65
current 133
curshape 65
Cursor 65
Cursorposition 63, 65
custom-remove 142
d* 34
d+ 34
d- 34
d. 66
d.r 67
d< 34
d= 34
d0= 34
dabs 34
Datei
anlegen 15
anmelden 15
Datei-Steuerblock 84
Dateivariable 84
Datenformat 66
Datenstrukturen 113
Datentyp 113
debug 178
decode 117
Defer 116, 124, 128
defining word 192
Definition 6, 188
definitions 133
Dekompiler 183
del 64
delete 90
depth 38
detract 79, 120
Dictionary 136, 188
Dictionarypointer 68
digit? 81
dir 90
direct 87, 88, 105
Directory 85
Direktzugriff 87
display 73, 118, 187
Division 188
dnegate 34
DO 42
document 25
Does> 121, 156
dos 89
DOS-File 84
dos: 90
DP 68, 110, 140
dpl 82
drive 92
drop 36
Drucker 25, 66

Druckerspooler 165
Druckersteuerung 66
drv 92
ds@ 111
DTA 92
Dummy 49
dump 111, 183
dup 36
Eaker-CASE 51
Editor 16
Start des Compilers 21
Tastenbelegung 18
ELSE 42
ELSECASE 52
emit 73
empty 142
empty-buffers 97
empty-keys 68
emptybuf 97
end-code 16Ø
END-TRACE 175, 185
ENDIF 42, 125
endloop 44, 179
Entwicklungssystem 1Ø7
eof 96
erase 1Ø9
error condition 189
error" 46
error# 1Ø6
errorhandler 47, 127
erweiterten Adreßraum 1Ø7
Escape 62
even 3Ø
execute 45, 72, 129
execution vector 58
exists? 41, 151
exit 41, 44, 121
expect 7Ø, 117
extend 33
Extension 86
F83-NUMBER? 82
Faktorisierung 174
Fakultät 59
Fallunterscheidung 47
false 32
fblock! 96
fblock@ 96
FCB 84
fclose 1Ø1
Fehlercode 1Ø7
Fehlersuche 174
FELD
fgetc 1Ø1
FIFO 12Ø
figFORTH 9
file 92

file! 1Ø2
File-Interface 84
file-link 1Ø3
file? 93
file@ 1Ø2
filename 92
files 15, 26, 86, 91
fill 1Ø9
find 71, 8Ø, 154
first 98
fix 17, 26
fkey 59
Flag 189
flip 1Ø9
Floor 189
flush 87, 97, 1Ø1
fnamelen 92
fopen 1Ø1
forget 141, 172
Forth 134
forth-83 134
FORTH-File 84
FORTH-Screen 16
fputc 1Ø2
freebuffer 98
freset 1Ø1
from 1ØØ
fromfile 16, 91
fsearch 1Ø4
fseek 1Ø2
fswap 91
ftype 9Ø
full 2Ø, 26, 63
Funktionstaste 59
Funktionstasten 69
get 121
global 35
Glossar
- 32Bit-Worte 33
- Arithmetische Funktionen 29
- Assembler 16Ø
- Ausdrucken von Screens 25
- Debugging 185
- Kontrollstrukturen 41
- Logik und Vergleiche 32
- Multitasker 169
- Speicheroperationen 1Ø7
- Stackoperationen 36
- Terminal-I/O 62
- Vektorisierung 128
- Vokabular 133

glossary 189
GOTOXY 125
halign 145
hallot 145
Handle-Nummer 84

Handlenummer 91
have 41
headerless words 144
Heap 141, 144, 145
heap? 145
help 16, 26
here 1Ø9, 14Ø
hide 143
high-Byte 1Ø9
high-word 1Ø7
hold 83
I 43
IF 42
immediate 155
immediate word 192
immediate-Bit 137
in-line code 16Ø
include 27, 94
index 26, 99
indirect-Bit 137
input 68, 127
input stream 19Ø
input# 48, 82
Input: 117, 127, 129
inputkol 63
Instructionpointer 146, 176
Instruktionszeiger 146
INTEL-Prozessor 1Ø7
interpret 72, 152
Interpreter 153, 189
IP 146, 176
Is 116, 124, 128
is-depth 46
isdepth 181
isfile 16, 91
isfile@ 91
J 43
Kaltstartwerte 143
kernel 123
key 69, 117
key? 68, 117
keyboard 68, 117, 129
killfile 1Ø1
Kommandozeile 177
Kommentare 86
Kompilationsadresse 136, 176, 185
Kompilationsvokabular 133
Konsolentask 165
Kontrollstrukturen 189
l! 111
l/s 62
l>name 143
l@ 111
Label 161
lallocate 112
last 59, 136, 142

last' 60
LATEST 59
lc! 111
lc@ 111
ldump 112, 183
leave 43
Leerzeichen 62
lfa 136
lfgets 102
lfputs 102
lfree 112
lfsave 102
limit 98
Linefeed 62
link 136, 137
Linkfeld 132, 137
link> 143
list 26, 64, 91
Liste 132
listing 25
Literal 155
Literatur
    Allgemeines über FORTH 27
    Kontrollstrukturen 59
lmove 111
load 93, 150, 189
loadfile 103, 152
loadfrom 94
loadscreen 85
lock 169
lokal 36
LOOP 43
Low-Byte 109
low-word 107
lst! 66
ltype 74, 112
m* 34
m/mod 34
make 93
makefile 86, 93
makeview 131
Makro 160
mass storage 190
match 79
Matrix 119
max 30
md 85, 90
MetaCompilation 123
Method: 121
Methods> 121
min 30
mod 31
Module 165
more 86, 94
move 108
msdos 88

MSDOS-Fileinterface 88
multitask 165, 169
Multitasking 78, 165
myself 60
n>link 143
name 71, 136, 138, 141, 153
name> 143
Namensfeld 138
negate 30
nest 179
NEXT 146
next-link 185
nfa 136
nip 37
noop 72, 124, 129
not 30
notfound 154
Null-Byte 80
nullstring? 70, 76
number 82
number? 81, 82
off 109, 122
offset 96, 98, 107
Offsetadresse 107
on 109
Only 134
Onlyforth 134
open 101
Operator 66
Operatoren 121
or 33
order 15, 26
origin 143
out of range 57
output 73
Output: 118, 130
outputkol 63
over 37
PAD 67, 68, 110
page 20, 26, 64
Parameter 36, 136, 138
Parameterfeld 117, 138, 139, 146
parse 71, 153
parser 153
pass 169
patch 139
patchen 123
path 15, 26, 89
PAUSE 64, 69, 73, 170
pc! 67
pc@ 67
perform 45, 72, 121, 124, 127, 129
Peripheriebaustein 67
pfa 136
Pfad 85
pick 38

place 78, 11Ø
plist 25
port 67
positional CASE 56
Postfix notation 7
prev 98
print 66
printer 66
Programm 6, 113, 19Ø
  Befehlsteil 113
  Datenteil 113
prompt 153
Prozedur 6
Prozedurvariable 72
Prozesse 165
pthru 25
Puffer 12Ø
push 39
pushfile 1ØØ
put 121
query 72, 82
quit 47, 154
r# 26, 47
r/w 96
r> 38
r@ 39
rØ 39
range 99
rd 9Ø
rdepth 38
rdrop 39
receive 188
recurse 6Ø
recursive 59, 157
Register 158
Registerbelegung 158
Registeroperationen 159
Rekursion 59, 19Ø
remove 142
ren 9Ø
rendezvous 17Ø
REPEAT 45
restart 152, 177, 185
restorevideo 64
restrict 155
restrict-Bit 137
return to caller 41
Return-Taste 62
RETURN_CODE 1Ø7
Returnstack 39, 47, 191
Returnstackpointer 146
reveal 143
Ringpuffer 12Ø
roll 37
rot 37
Routine 6

row 65
RP 146
rp! 39
rp@ 39
runtime library 22, 123
s>d 33
sØ 38
save 63, 142
save-buffers 97
savefile 1Ø2
savesystem 88
savevideo 64
scan 77
Scancode 69
Schleifenindex 42, 43
Schleifenrumpf 42
Schleifenvariable 119
Schrittweite 43
scr 25, 47
Screen 19Ø
screen files 87
Screen-Files 85
seal 134
search 79
search order 191
see 183
SEGMENT 1Ø7
Segmentadresse 1Ø7
Segmentregister 1Ø7
Semaphore 167
setpage 63
Shadow-Konzept 16
Shadow-Screens 86
show 121
Showload 21
sign 83
singletask 17Ø
skip 77
sleep 17Ø
source 71, 15Ø
SP 146
sp! 38
sp@ 38
space 74
spaces 74
Spaltennummer 65
span 7Ø
Speicheradressen 1Ø7
Stack 28, 35, 36, 18Ø, 191
Stacknotation 28
Stackoperationen 35
Stackpointer 146
Standard-Prolog 125
standardi/o 63
state 15Ø
status 14, 26

Steueranweisungen 40
Steuerzeichen 69
stop 171
stop? 46, 70
String 70, 75, 191
string literal 76
Stringvariable 76
Stringvergleich 76
Suchreihenfolge 132
swap 37
Synonymdeklaration 119
Synonyme 188
Systemverhalten
    anpassen 123
    beeinflussen 124
    umschalten 124
Tabelle 56
Task 165, 171
Taskwechsler 165
Tastendruck 69
Teilstrings 77
terminal 63
Text-Interpreter 72
THEN 42
thru 151
tib 67
tipp 73
toss 115, 134
TRACE 175, 176
true 32, 192
type 73, 78
Typisierung 113
u. 66
u.r 67
u/mod 31
u< 33
u> 33
uallot 110, 140
ud/mod 35
udp 110, 140
um* 34
um/mod 35
umax 32
umin 32
under 37
UNDO 44
unlock 171
unnest 41, 179
unravel 180
UNTIL 45
up! 172
up@ 172
update 97
upper 77
use 86
User 118, 172

Userarea 192
Uservariable 118, 142, 192
uwithin 33
Variable 114
Vektor 55
vektorielle Programmausführung 124
Vektorisierung 61
Verlassen des volksFORTH 152
video@ 63
Videokarte 63
view 16, 26, 91
voc-link 135
Vocabulary 117, 133, 192
vocs 15, 26
Vokabular 132
Vokabular-Konzept 8
Vorwärts-Referenzen 124, 128
vp 135
W 146
wake 172
WHILE 45
Window 63
word 71, 153
words 15, 26, 134
Wortregister 146
Worttypen 138
xor 33, 109
Zahleneingabe 48, 82
Zähler 165
Zeichenattribut 64
Zeichenkette 75
Zeichenketten 148
Zeilenende 112
Zeilennummer 65
Zugriffsmethode 121